Introduction

Welcome to Move, a next generation language for secure, sandboxed, and formally verified programming. Its first use case is for the Diem blockchain, where Move provides the foundation for its implementation. Move allows developers to write programs that flexibly manage and transfer assets, while providing the security and protections against attacks on those assets. However, Move has been developed with use cases in mind outside a blockchain context as well.

Move takes its cue from Rust by using resource types with move (hence the name) semantics as an explicit representation of digital assets, such as currency.

Who is Move for?

Move was designed and created as a secure, verified, yet flexible programming language. The first use of Move is for the implementation of the Diem blockchain. That said, the language is still evolving. Move has the potential to be a language for other blockchains, and even non-blockchain use cases as well.

The early Move Developer is one with some programming experience, who wants to begin understanding the core programming language and see examples of its usage.

Hobbyists

Understanding that the capability to create custom modules on the Diem Payment Network will not be available at launch, the hobbyist Move Developer is interested in learning the intricacies of the language. She will understand the basic syntax, the standard libraries available, and write example

code that can be executed using the Move CLI. The Move Developer may even want to dig into understanding how the Move Virtual Machine executes the code she writes.

Core Contributor

Beyond a hobbyist wanting to stay ahead of the curve for the core programming language is someone who may want to contribute directly to Move. Whether this includes submitting language improvements or even, in the future, adding core modules available on the Diem Payment Network, the core contributor will understand Move at a deep level.

Who Move is currently not targeting

Currently, Move is not targeting developers who wish to create custom modules and contracts for use on the Diem Payment Network. We are also not targeting novice developers who expect a completely polished developer experience even in testing the language.

Where Do I Start?

Begin with understanding modules and scripts and then work through the Move Tutorial.

Modules and Scripts

Move has two different types of programs: ***Modules*** and ***Scripts***. Modules are libraries that define struct types along with functions that operate on these types. Struct types define the schema of Move's global storage, and module functions define the rules for updating storage. Modules themselves are also stored in global storage. Scripts are executable entrypoints similar to a `main` function in a conventional language. A script typically calls functions of a published module that perform updates to global storage. Scripts are ephemeral code snippets that are not published in global storage.

A Move source file (or **compilation unit**) may contain multiple modules and scripts. However, publishing a module or executing a script are separate VM operations.

Syntax

Scripts

A script has the following structure:

```
script {
    <use>*
    <constants>*
    fun <identifier><[type parameters: constraint]*>([identifier: type]*)
<function_body>
}
```

A `script` block must start with all of its `use` declarations, followed by any constants and (finally) the main function declaration. The main function can have any name (i.e., it need not be called `main`), is the only function in a script block, can have any number of arguments, and must not return a value. Here is an example with each of these components:

```
script {
    // Import the debug module published at the named account address std.
    use std::debug;

    const ONE: u64 = 1;

    fun main(x: u64) {
        let sum = x + ONE;
        debug::print(&sum)
    }
}
```

Scripts have very limited power—they cannot declare friends, struct types or access global storage. Their primary purpose is to invoke module functions.

Modules

A module has the following syntax:

```
module <address>::<identifier> {
    (<use> | <friend> | <type> | <function> | <constant>)*
}
```

where `<address>` is a valid named or literal address.

For example:

```
module 0x42::test {
    struct Example has copy, drop { i: u64 }

    use std::debug;
    friend 0x42::another_test;

    const ONE: u64 = 1;

    public fun print(x: u64) {
        let sum = x + ONE;
        let example = Example { i: sum };
        debug::print(&sum)
    }
}
```

The `module 0x42::test` part specifies that the module `test` will be published under the account address `0x42` in global storage.

Modules can also be declared using named addresses. For example:

```
module test_addr::test {
    struct Example has copy, drop { a: address }

    use std::debug;
    friend test_addr::another_test;

    public fun print() {
        let example = Example { a: @test_addr };
        debug::print(&example)
    }
}
```

Because named addresses only exist at the source language level and during compilation, named addresses will be fully substituted for their value at the bytecode level. For example if we had the following code:

```
script {
    fun example() {
        my_addr::m::foo(@my_addr);
    }
}
```

and we compiled it with `my_addr` set to `0xC0FFEE`, then it would be equivalent to the following operationally:

```
script {
    fun example() {
        0xC0FFEE::m::foo(@0xC0FFEE);
    }
}
```

However at the source level, these *are not equivalent*—the function `m::foo` *must* be accessed through the `my_addr` named address, and not through the numerical value assigned to that address.

Module names can start with letters `a` to `z` or letters `A` to `Z`. After the first character, module names can contain underscores `_`, letters `a` to `z`, letters `A` to `Z`, or digits `0` to `9`.

```
module my_module {}
module foo_bar_42 {}
```

Typically, module names start with an lowercase letter. A module named `my_module` should be stored in a source file named `my_module.move`.

All elements inside a `module` block can appear in any order. Fundamentally, a module is a collection of `types` and `functions`. The `use` keyword is used to import types from other modules. The `friend` keyword specifies a list of trusted modules. The `const` keyword defines private constants that can be used in the functions of a module.

Move Tutorial

Please refer to the Move Tutorial.

Integers

Move supports six unsigned integer types: `u8`, `u16`, `u32`, `u64`, `u128`, and `u256`. Values of these types range from 0 to a maximum that depends on the size of the type.

Type	Value Range
Unsigned 8-bit integer, `u8`	0 to $2^8 - 1$
Unsigned 16-bit integer, `u16`	0 to $2^{16} - 1$
Unsigned 32-bit integer, `u32`	0 to $2^{32} - 1$
Unsigned 64-bit integer, `u64`	0 to $2^{64} - 1$
Unsigned 128-bit integer, `u128`	0 to $2^{128} - 1$
Unsigned 256-bit integer, `u256`	0 to $2^{256} - 1$

Literals

Literal values for these types are specified either as a sequence of digits (e.g., `112`) or as hex literals, e.g., `0xFF`. The type of the literal can optionally be added as a suffix, e.g., `112u8`. If the type is not specified, the compiler will try to infer the type from the context where the literal is used. If the type cannot be inferred, it is assumed to be `u64`.

Number literals can be separated by underscores for grouping and readability. (e.g., `1_234_5678`, `1_000u128`, `0xAB_CD_12_35`).

If a literal is too large for its specified (or inferred) size range, an error is reported.

Examples

```
// literals with explicit annotations;
let explicit_u8 = 1u8;
let explicit_u16 = 1u16;
let explicit_u32 = 1u32;
let explicit_u64 = 2u64;
let explicit_u128 = 3u128;
let explicit_u256 = 1u256;
let explicit_u64_underscored = 154_322_973u64;

// literals with simple inference
let simple_u8: u8 = 1;
let simple_u16: u16 = 1;
let simple_u32: u32 = 1;
let simple_u64: u64 = 2;
let simple_u128: u128 = 3;
let simple_u256: u256 = 1;

// literals with more complex inference
let complex_u8 = 1; // inferred: u8
// right hand argument to shift must be u8
let _unused = 10 << complex_u8;

let x: u8 = 38;
let complex_u8 = 2; // inferred: u8
// arguments to `+` must have the same type
let _unused = x + complex_u8;

let complex_u128 = 133_876; // inferred: u128
// inferred from function argument type
function_that_takes_u128(complex_u128);

// literals can be written in hex
let hex_u8: u8 = 0x1;
let hex_u16: u16 = 0x1BAE;
let hex_u32: u32 = 0xDEAD80;
let hex_u64: u64 = 0xCAFE;
```

```
let hex_u128: u128 = 0xDEADBEEF;
let hex_u256: u256 = 0x1123_456A_BCDE_F;
```

Operations

Arithmetic

Each of these types supports the same set of checked arithmetic operations. For all of these operations, both arguments (the left and right side operands) *must* be of the same type. If you need to operate over values of different types, you will need to first perform a cast. Similarly, if you expect the result of the operation to be too large for the integer type, perform a cast to a larger size before performing the operation.

All arithmetic operations abort instead of behaving in a way that mathematical integers would not (e.g., overflow, underflow, divide-by-zero).

Syntax	Operation	Aborts If
`+`	addition	Result is too large for the integer type
`-`	subtraction	Result is less than zero
`*`	multiplication	Result is too large for the integer type
`%`	modular division	The divisor is `0`
`/`	truncating division	The divisor is `0`

Bitwise

The integer types support the following bitwise operations that treat each number as a series of individual bits, either 0 or 1, instead of as numerical integer values.

Bitwise operations do not abort.

Syntax	Operation	Description
`&`	bitwise and	Performs a boolean and for each bit pairwise
`|`	bitwise or	Performs a boolean or for each bit pairwise
`^`	bitwise xor	Performs a boolean exclusive or for each bit pairwise

Bit Shifts

Similar to the bitwise operations, each integer type supports bit shifts. But unlike the other operations, the righthand side operand (how many bits to shift by) must *always* be a `u8` and need not match the left side operand (the number you are shifting).

Bit shifts can abort if the number of bits to shift by is greater than or equal to `8`, `16`, `32`, `64`, `128` or `256` for `u8`, `u16`, `u32`, `u64`, `u128` and `u256` respectively.

Syntax	Operation	Aborts if
`<<`	shift left	Number of bits to shift by is greater than the size of the integer type
`>>`	shift right	Number of bits to shift by is greater than the size of the integer type

Comparisons

Integer types are the *only* types in Move that can use the comparison operators. Both arguments need to be of the same type. If you need to compare integers of different types, you will need to cast one of them first.

Comparison operations do not abort.

Syntax	Operation
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

Equality

Like all types with `drop` in Move, all integer types support the "equal" and "not equal" operations. Both arguments need to be of the same type. If you need to compare integers of different types, you will need to cast one of them first.

Equality operations do not abort.

Syntax	Operation
`==`	equal
`!=`	not equal

For more details see the section on equality

Casting

Integer types of one size can be cast to integer types of another size. Integers are the only types in Move that support casting.

Casts *do not* truncate. Casting will abort if the result is too large for the specified type

Syntax	Operation	Aborts if
`(e as T)`	Cast integer expression `e` into an integer type `T`	`e` is too large to represent as a `T`

Here, the type of `e` must be `8`, `16`, `32`, `64`, `128` or `256` and `T` must be `u8`, `u16`, `u32`, `u64`, `u128` oe `u256`.

For example:

- `(x as u8)`
- `(y as u16)`
- `(873u16 as u32)`
- `(2u8 as u64)`
- `(1 + 3 as u128)`
- `(4/2 + 12345 as u256)`

Ownership

As with the other scalar values built-in to the language, integer values are implicitly copyable, meaning they can be copied without an explicit instruction such as `copy`.

Bool

`bool` is Move's primitive type for boolean `true` and `false` values.

Literals

Literals for `bool` are either `true` or `false`.

Operations

Logical

`bool` supports three logical operations:

Syntax	Description	Equivalent Expression
`&&`	short-circuiting logical and	`p && q` is equivalent to `if (p) q else false`
`||`	short-circuiting logical or	`p || q` is equivalent to `if (p) true else q`
`!`	logical negation	`!p` is equivalent to `if (p) false else true`

Control Flow

`bool` values are used in several of Move's control-flow constructs:

- `if (bool) { ... }`
- `while (bool) { .. }`
- `assert!(bool, u64)`

Ownership

As with the other scalar values built-in to the language, boolean values are implicitly copyable, meaning they can be copied without an explicit instruction such as `copy`.

Address

`address` is a built-in type in Move that is used to represent locations (sometimes called accounts) in global storage. An `address` value is a 128-bit (16 byte) identifier. At a given address, two things can be stored: Modules and Resources.

Although an `address` is a 128 bit integer under the hood, Move addresses are intentionally opaque---they cannot be created from integers, they do not support arithmetic operations, and they cannot be modified. Even though there might be interesting programs that would use such a feature (e.g., pointer arithmetic in C fills a similar niche), Move does not allow this dynamic behavior because it has been designed from the ground up to support static verification.

You can use runtime address values (values of type `address`) to access resources at that address. You *cannot* access modules at runtime via address values.

Addresses and Their Syntax

Addresses come in two flavors, named or numerical. The syntax for a named address follows the same rules for any named identifier in Move. The syntax of a numerical address is not restricted to hex-encoded values, and any valid `u128` numerical value can be used as an address value, e.g., `42`, `0xCAFE`, and `2021` are all valid numerical address literals.

To distinguish when an address is being used in an expression context or not, the syntax when using an address differs depending on the context where it's used:

- When an address is used as an expression the address must be prefixed by the `@` character, i.e., `@<numerical_value>` or `@<named_address_identifier>`.

- Outside of expression contexts, the address may be written without the leading `@` character, i.e., `<numerical_value>` or `<named_address_identifier>`.

In general, you can think of `@` as an operator that takes an address from being a namespace item to being an expression item.

Named Addresses

Named addresses are a feature that allow identifiers to be used in place of numerical values in any spot where addresses are used, and not just at the value level. Named addresses are declared and bound as top level elements (outside of modules and scripts) in Move Packages, or passed as arguments to the Move compiler.

Named addresses only exist at the source language level and will be fully substituted for their value at the bytecode level. Because of this, modules and module members *must* be accessed through the module's named address and not through the numerical value assigned to the named address during compilation, e.g., `use my_addr::foo` is *not* equivalent to `use 0x2::foo` even if the Move program is compiled with `my_addr` set to `0x2`. This distinction is discussed in more detail in the section on Modules and Scripts.

Examples

```
let a1: address = @0x1; // shorthand for 0x00000000000000000000000000000001
let a2: address = @0x42; // shorthand for 0x00000000000000000000000000000042
let a3: address = @0xDEADBEEF; // shorthand for 0x00000000000000000000000DEADBEEF
let a4: address = @0x0000000000000000000000000000000A;
let a5: address = @std; // Assigns `a5` the value of the named address `std`
let a6: address = @66;
let a7: address = @0x42;

module 66::some_module {   // Not in expression context, so no @ needed
    use 0x1::other_module; // Not in expression context so no @ needed
    use std::vector;       // Can use a named address as a namespace item when using
other modules
    ...
}

module std::other_module {  // Can use a named address as a namespace item to declare a
module
    ...
}
```

Global Storage Operations

The primary purpose of `address` values are to interact with the global storage operations.

`address` values are used with the `exists`, `borrow_global`, `borrow_global_mut`, and `move_from` operations.

The only global storage operation that *does not* use `address` is `move_to`, which uses `signer`.

Ownership

As with the other scalar values built-in to the language, `address` values are implicitly copyable, meaning they can be copied without an explicit instruction such as `copy`.

Vector

`vector<T>` is the only primitive collection type provided by Move. A `vector<T>` is a homogenous collection of `T`'s that can grow or shrink by pushing/popping values off the "end".

A `vector<T>` can be instantiated with any type `T`. For example, `vector<u64>`, `vector<address>`, `vector<0x42::MyModule::MyResource>`, and `vector<vector<u8>>` are all valid vector types.

Literals

General `vector` Literals

Vectors of any type can be created with `vector` literals.

Syntax	Type	Description
`vector[]`	`vector[]: vector<T>` where `T` is any single, non-reference type	An empty vector
`vector[e1, ..., en]`	`vector[e1, ..., en]: vector<T>` where `e_i: T` s.t. `0 < i <= n` and `n > 0`	A vector with `n` elements (of length `n`)

In these cases, the type of the `vector` is inferred, either from the element type or from the vector's usage. If the type cannot be inferred, or simply for added clarity, the type can be specified explicitly:

```
vector<T>[]: vector<T>
vector<T>[e1, ..., en]: vector<T>
```

Example Vector Literals

```
(vector[]: vector<bool>);
(vector[0u8, 1u8, 2u8]: vector<u8>);
(vector<u128>[]: vector<u128>);
(vector<address>[@0x42, @0x100]: vector<address>);
```

`vector<u8>` literals

A common use-case for vectors in Move is to represent "byte arrays", which are represented with `vector<u8>`. These values are often used for cryptographic purposes, such as a public key or a hash result. These values are so common that specific syntax is provided to make the values more readable, as opposed to having to use `vector[]` where each individual `u8` value is specified in numeric form.

There are currently two supported types of `vector<u8>` literals, *byte strings* and *hex strings*.

Byte Strings

Byte strings are quoted string literals prefixed by a `b`, e.g. `b"Hello!\n"`.

These are ASCII encoded strings that allow for escape sequences. Currently, the supported escape sequences are:

Escape Sequence	Description
`\n`	New line (or Line feed)
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash

Escape Sequence	Description
`\0`	Null
`\"`	Quote
`\xHH`	Hex escape, inserts the hex byte sequence `HH`

Hex Strings

Hex strings are quoted string literals prefixed by a `x`, e.g. `x"48656C6C6F210A"`.

Each byte pair, ranging from `00` to `FF`, is interpreted as hex encoded `u8` value. So each byte pair corresponds to a single entry in the resulting `vector<u8>`.

Example String Literals

```
script {
fun byte_and_hex_strings() {
    assert!(b"" == x"", 0);
    assert!(b"Hello!\n" == x"48656C6C6F210A", 1);
    assert!(b"\x48\x65\x6C\x6C\x6F\x21\x0A" == x"48656C6C6F210A", 2);
    assert!(
        b"\"Hello\tworld!\"\n \r \\Null=\0" ==
            x"2248656C6C6F09776F726C6421220A200D205C4E756C6C3D00",
        3
    );
}
}
```

Operations

`vector` supports the following operations via the `std::vector` module in the Move standard library:

Function	Description	Aborts?
`vector::empty<T>(): vector<T>`	Create an empty vector that can store values of type `T`	Never
`vector::singleton<T>(t: T): vector<T>`	Create a vector of size 1 containing `t`	Never
`vector::push_back<T>(v: &mut vector<T>, t: T)`	Add `t` to the end of `v`	Never
`vector::pop_back<T>(v: &mut vector<T>): T`	Remove and return the last element in `v`	If `v` is empty
`vector::borrow<T>(v: &vector<T>, i: u64): &T`	Return an immutable reference to the `T` at index `i`	If `i` is not in bounds
`vector::borrow_mut<T>(v: &mut vector<T>, i: u64): &mut T`	Return a mutable reference to the `T` at index `i`	If `i` is not in bounds
`vector::destroy_empty<T>(v: vector<T>)`	Delete `v`	If `v` is not empty
`vector::append<T>(v1: &mut vector<T>, v2: vector<T>)`	Add the elements in `v2` to the end of `v1`	Never
`vector::contains<T>(v: &vector<T>, e: &T): bool`	Return true if `e` is in the vector `v`. Otherwise, returns false	Never
`vector::swap<T>(v: &mut vector<T>, i: u64, j: u64)`	Swaps the elements at the `i` th and `j` th indices in the vector `v`	If `i` or `j` is out of bounds

Function	Description	Aborts?
`vector::reverse<T>(v: &mut vector<T>)`	Reverses the order of the elements in the vector `v` in place	Never
`vector::index_of<T>(v: &vector<T>, e: &T): (bool, u64)`	Return `(true, i)` if `e` is in the vector `v` at index `i`. Otherwise, returns `(false, 0)`	Never
`vector::remove<T>(v: &mut vector<T>, i: u64): T`	Remove the `i`th element of the vector `v`, shifting all subsequent elements. This is O(n) and preserves ordering of elements in the vector	If `i` is out of bounds
`vector::swap_remove<T>(v: &mut vector<T>, i: u64): T`	Swap the `i`th element of the vector `v` with the last element and then pop the element, This is O(1), but does not preserve ordering of elements in the vector	If `i` is out of bounds

More operations may be added over time.

Example

```
use std::vector;

let v = vector::empty<u64>();
vector::push_back(&mut v, 5);
vector::push_back(&mut v, 6);

assert!(*vector::borrow(&v, 0) == 5, 42);
assert!(*vector::borrow(&v, 1) == 6, 42);
assert!(vector::pop_back(&mut v) == 6, 42);
assert!(vector::pop_back(&mut v) == 5, 42);
```

Destroying and copying `vectors`

Some behaviors of `vector<T>` depend on the abilities of the element type, `T`. For example, vectors containing elements that do not have `drop` cannot be implicitly discarded like `v` in the example above--they must be explicitly destroyed with `vector::destroy_empty`.

Note that `vector::destroy_empty` will abort at runtime unless `vec` contains zero elements:

```
fun destroy_any_vector<T>(vec: vector<T>) {
    vector::destroy_empty(vec) // deleting this line will cause a compiler error
}
```

But no error would occur for dropping a vector that contains elements with `drop`:

```
fun destroy_droppable_vector<T: drop>(vec: vector<T>) {
    // valid!
    // nothing needs to be done explicitly to destroy the vector
}
```

Similarly, vectors cannot be copied unless the element type has `copy`. In other words, a `vector<T>` has `copy` if and only if `T` has `copy`. However, even copyable vectors are never implicitly copied:

```
let x = vector::singleton<u64>(10);
let y = copy x; // compiler error without the copy!
```

Copies of large vectors can be expensive, so the compiler requires explicit `copy`'s to make it easier to see where they are happening.

For more details see the sections on type abilities and generics.

Ownership

As mentioned above, `vector` values can be copied only if the elements can be copied. In that case, the copy must be explicit via a `copy` or a dereference `*`.

Signer

`signer` is a built-in Move resource type. A `signer` is a capability that allows the holder to act on behalf of a particular `address`. You can think of the native implementation as being:

```
struct signer has drop { a: address }
```

A `signer` is somewhat similar to a Unix UID in that it represents a user authenticated by code *outside* of Move (e.g., by checking a cryptographic signature or password).

Comparison to `address`

A Move program can create any `address` value without special permission using address literals:

```
let a1 = @0x1;
let a2 = @0x2;
// ... and so on for every other possible address
```

However, `signer` values are special because they cannot be created via literals or instructions--only by the Move VM. Before the VM runs a script with parameters of type `signer`, it will automatically create `signer` values and pass them into the script:

```
script {
    use std::signer;
    fun main(s: signer) {
        assert!(signer::address_of(&s) == @0x42, 0);
    }
}
```

This script will abort with code `0` if the script is sent from any address other than `0x42`.

A transaction script can have an arbitrary number of `signer`s as long as the `signer`s are a prefix to any other arguments. In other words, all of the `signer` arguments must come first:

```
script {
    use std::signer;
    fun main(s1: signer, s2: signer, x: u64, y: u8) {
        // ...
    }
}
```

This is useful for implementing *multi-signer scripts* that atomically act with the authority of multiple parties. For example, an extension of the script above could perform an atomic currency swap between `s1` and `s2`.

`signer` Operators

The `std::signer` standard library module provides two utility functions over `signer` values:

Function	Description
`signer::address_of(&signer): address`	Return the `address` wrapped by this `&signer`.
`signer::borrow_address(&signer): &address`	Return a reference to the `address` wrapped by this `&signer`.

In addition, the `move_to<T>(&signer, T)` global storage operator requires a `&signer` argument to publish a resource `T` under `signer.address`'s account. This ensures that only an authenticated user can elect to publish a resource under their `address`.

Ownership

Unlike simple scalar values, `signer` values are not copyable, meaning they cannot be copied (from any operation whether it be through an explicit `copy` instruction or through a dereference `*`.

References

Move has two types of references: immutable `&` and mutable `&mut`. Immutable references are read only, and cannot modify the underlying value (or any of its fields). Mutable references allow for modifications via a write through that reference. Move's type system enforces an ownership discipline that prevents reference errors.

For more details on the rules of references, see Structs and Resources

Reference Operators

Move provides operators for creating and extending references as well as converting a mutable reference to an immutable one. Here and elsewhere, we use the notation `e: T` for "expression `e` has type `T`".

Syntax	Type	Description
`&e`	`&T` where `e: T` and `T` is a non-reference type	Create an immutable reference to `e`
`&mut e`	`&mut T` where `e: T` and `T` is a non-reference type	Create a mutable reference to `e`.
`&e.f`	`&T` where `e.f: T`	Create an immutable reference to field `f` of struct `e`.
`&mut e.f`	`&mut T` where `e.f: T`	Create a mutable reference to field `f` of struct `e`.
`freeze(e)`	`&T` where `e: &mut T`	Convert the mutable reference `e` into an immutable reference.

The `&e.f` and `&mut e.f` operators can be used both to create a new reference into a struct or to extend an existing reference:

```
let s = S { f: 10 };
let f_ref1: &u64 = &s.f; // works
let s_ref: &S = &s;
let f_ref2: &u64 = &s_ref.f // also works
```

A reference expression with multiple fields works as long as both structs are in the same module:

```
struct A { b: B }
struct B { c : u64 }
fun f(a: &A): &u64 {
  &a.b.c
}
```

Finally, note that references to references are not allowed:

```
let x = 7;
let y: &u64 = &x;
let z: &&u64 = &y; // will not compile
```

Reading and Writing Through References

Both mutable and immutable references can be read to produce a copy of the referenced value.

Only mutable references can be written. A write `*x = v` discards the value previously stored in `x` and updates it with `v`.

Both operations use the C-like `*` syntax. However, note that a read is an expression, whereas a write is a mutation that must occur on the left hand side of an equals.

Syntax	Type	Description
`*e`	`T` where `e` is `&T` or `&mut T`	Read the value pointed to by `e`
`*e1 = e2`	`()` where `e1: &mut T` and `e2: T`	Update the value in `e1` with `e2`.

In order for a reference to be read, the underlying type must have the `copy` ability as reading the reference creates a new copy of the value. This rule prevents the copying of resource values:

```
fun copy_resource_via_ref_bad(c: Coin) {
    let c_ref = &c;
    let counterfeit: Coin = *c_ref; // not allowed!
    pay(c);
    pay(counterfeit);
}
```

Dually: in order for a reference to be written to, the underlying type must have the `drop` ability as writing to the reference will discard (or "drop") the old value. This rule prevents the destruction of resource values:

```
fun destroy_resource_via_ref_bad(ten_coins: Coin, c: Coin) {
    let ref = &mut ten_coins;
    *ref = c; // not allowed--would destroy 10 coins!
}
```

`freeze` inference

A mutable reference can be used in a context where an immutable reference is expected:

```
let x = 7;
let y: &u64 = &mut x;
```

This works because the under the hood, the compiler inserts `freeze` instructions where they are needed. Here are a few more examples of `freeze` inference in action:

```
fun takes_immut_returns_immut(x: &u64): &u64 { x }

// freeze inference on return value
fun takes_mut_returns_immut(x: &mut u64): &u64 { x }

fun expression_examples() {
    let x = 0;
    let y = 0;
    takes_immut_returns_immut(&x); // no inference
    takes_immut_returns_immut(&mut x); // inferred freeze(&mut x)
    takes_mut_returns_immut(&mut x); // no inference

    assert!(&x == &mut y, 42); // inferred freeze(&mut y)
}

fun assignment_examples() {
    let x = 0;
    let y = 0;
    let imm_ref: &u64 = &x;

    imm_ref = &x; // no inference
    imm_ref = &mut y; // inferred freeze(&mut y)
}
```

Subtyping

With this `freeze` inference, the Move type checker can view `&mut T` as a subtype of `&T`. As shown above, this means that anywhere for any expression where a `&T` value is used, a `&mut T` value can also be used. This terminology is used in error messages to concisely indicate that a `&mut T` was needed where a `&T` was supplied. For example

```
address 0x42 {
module example {
    fun read_and_assign(store: &mut u64, new_value: &u64) {
        *store = *new_value
    }

    fun subtype_examples() {
        let x: &u64 = &0;
        let y: &mut u64 = &mut 1;

        x = &mut 1; // valid
        y = &2; // invalid!

        read_and_assign(y, x); // valid
        read_and_assign(x, y); // invalid!
    }
}
}
```

will yield the following error messages

```
error:

    ┌── example.move:12:9 ───
    │
 12 │         y = &2; // invalid!
    │         ^ Invalid assignment to local 'y'
    ·
 12 │         y = &2; // invalid!
    │             -- The type: '&{integer}'
    ·
  9 │         let y: &mut u64 = &mut 1;
    │                --------- Is not a subtype of: '&mut u64'
    │

error:

    ┌── example.move:15:9 ───
    │
 15 │         read_and_assign(x, y); // invalid!
    │         ^^^^^^^^^^^^^^^^^^^^ Invalid call of '0x42::example::read_and_assign'.
Invalid argument for parameter 'store'
    ·
  8 │         let x: &u64 = &0;
    │                ---- The type: '&u64'
    ·
  3 │     fun read_and_assign(store: &mut u64, new_value: &u64) {
    │                                --------- Is not a subtype of: '&mut u64'
    │
```

The only other types currently that has subtyping are tuples

Ownership

Both mutable and immutable references can always be copied and extended *even if there are existing copies or extensions of the same reference*:

```
fun reference_copies(s: &mut S) {
  let s_copy1 = s; // ok
  let s_extension = &mut s.f; // also ok
  let s_copy2 = s; // still ok
  ...
}
```

This might be surprising for programmers familiar with Rust's ownership system, which would reject the code above. Move's type system is more permissive in its treatment of copies, but equally strict in ensuring unique ownership of mutable references before writes.

References Cannot Be Stored

References and tuples are the *only* types that cannot be stored as a field value of structs, which also means that they cannot exist in global storage. All references created during program execution will be destroyed when a Move program terminates; they are entirely ephemeral. This invariant is also true for values of types without the `store` ability, but note that references and tuples go a step further by never being allowed in structs in the first place.

This is another difference between Move and Rust, which allows references to be stored inside of structs.

Currently, Move cannot support this because references cannot be serialized, but *every Move value must be serializable*. This requirement comes from Move's persistent global storage, which needs to serialize values to persist them across program executions. Structs can be written to global storage, and thus they must be serializable.

One could imagine a fancier, more expressive, type system that would allow references to be stored in structs *and* ban those structs from existing in global storage. We could perhaps allow references inside of structs that do not have the `store` ability, but that would not completely solve the problem: Move has a fairly complex system for tracking static reference safety, and this aspect of the

type system would also have to be extended to support storing references inside of structs. In short, Move's type system (particularly the aspects around reference safety) would have to expand to support stored references. But it is something we are keeping an eye on as the language evolves.

Tuples and Unit

Move does not fully support tuples as one might expect coming from another language with them as a first-class value. However, in order to support multiple return values, Move has tuple-like expressions. These expressions do not result in a concrete value at runtime (there are no tuples in the bytecode), and as a result they are very limited: they can only appear in expressions (usually in the return position for a function); they cannot be bound to local variables; they cannot be stored in structs; and tuple types cannot be used to instantiate generics.

Similarly, unit `()` is a type created by the Move source language in order to be expression based. The unit value `()` does not result in any runtime value. We can consider unit `()` to be an empty tuple, and any restrictions that apply to tuples also apply to unit.

It might feel weird to have tuples in the language at all given these restrictions. But one of the most common use cases for tuples in other languages is for functions to allow functions to return multiple values. Some languages work around this by forcing the users to write structs that contain the multiple return values. However in Move, you cannot put references inside of structs. This required Move to support multiple return values. These multiple return values are all pushed on the stack at the bytecode level. At the source level, these multiple return values are represented using tuples.

Literals

Tuples are created by a comma separated list of expressions inside of parentheses.

Syntax	Type	Description
`()`	`(): ()`	Unit, the empty tuple, or the tuple of arity 0

Syntax	Type	Description
`(e1, ..., en)`	`(e1, ..., en): (T1, ..., Tn)` where `e_i: Ti` s.t. `0 < i <= n` and `n > 0`	A `n`-tuple, a tuple of arity `n`, a tuple with `n` elements

Note that `(e)` does not have type `(e): (t)`, in other words there is no tuple with one element. If there is only a single element inside of the parentheses, the parentheses are only used for disambiguation and do not carry any other special meaning.

Sometimes, tuples with two elements are called "pairs" and tuples with three elements are called "triples."

Examples

```
address 0x42 {
module example {
    // all 3 of these functions are equivalent

    // when no return type is provided, it is assumed to be `()`
    fun returns_unit_1() { }

    // there is an implicit () value in empty expression blocks
    fun returns_unit_2(): () { }

    // explicit version of `returns_unit_1` and `returns_unit_2`
    fun returns_unit_3(): () { () }


    fun returns_3_values(): (u64, bool, address) {
        (0, false, @0x42)
    }
    fun returns_4_values(x: &u64): (&u64, u8, u128, vector<u8>) {
        (x, 0, 1, b"foobar")
    }
}
}
```

Operations

The only operation that can be done on tuples currently is destructuring.

Destructuring

For tuples of any size, they can be destructured in either a `let` binding or in an assignment.

For example:

```
address 0x42 {
module example {
    // all 3 of these functions are equivalent
    fun returns_unit() {}
    fun returns_2_values(): (bool, bool) { (true, false) }
    fun returns_4_values(x: &u64): (&u64, u8, u128, vector<u8>) { (x, 0, 1, b"foobar")
}

    fun examples(cond: bool) {
        let () = ();
        let (x, y): (u8, u64) = (0, 1);
        let (a, b, c, d) = (@0x0, 0, false, b"");

        () = ();
        (x, y) = if (cond) (1, 2) else (3, 4);
        (a, b, c, d) = (@0x1, 1, true, b"1");
    }

    fun examples_with_function_calls() {
        let () = returns_unit();
        let (x, y): (bool, bool) = returns_2_values();
        let (a, b, c, d) = returns_4_values(&0);

        () = returns_unit();
        (x, y) = returns_2_values();
        (a, b, c, d) = returns_4_values(&1);
    }
}
}
```

For more details, see Move Variables.

Subtyping

Along with references, tuples are the only types that have subtyping in Move. Tuples do have subtyping only in the sense that subtype with references (in a covariant way).

For example:

```
let x: &u64 = &0;
let y: &mut u64 = &mut 1;

// (&u64, &mut u64) is a subtype of (&u64, &u64)
// since &mut u64 is a subtype of &u64
let (a, b): (&u64, &u64) = (x, y);

// (&mut u64, &mut u64) is a subtype of (&u64, &u64)
// since &mut u64 is a subtype of &u64
let (c, d): (&u64, &u64) = (y, y);

// error! (&u64, &mut u64) is NOT a subtype of (&mut u64, &mut u64)
// since &u64 is NOT a subtype of &mut u64
let (e, f): (&mut u64, &mut u64) = (x, y);
```

Ownership

As mentioned above, tuple values don't really exist at runtime. And currently they cannot be stored into local variables because of this (but it is likely that this feature will come soon). As such, tuples can only be moved currently, as copying them would require putting them into a local variable first.

Local Variables and Scope

Local variables in Move are lexically (statically) scoped. New variables are introduced with the keyword `let`, which will shadow any previous local with the same name. Locals are mutable and can be updated both directly and via a mutable reference.

Declaring Local Variables

`let` bindings

Move programs use `let` to bind variable names to values:

```
let x = 1;
let y = x + x:
```

`let` can also be used without binding a value to the local.

```
let x;
```

The local can then be assigned a value later.

```
let x;
if (cond) {
  x = 1
} else {
  x = 0
}
```

This can be very helpful when trying to extract a value from a loop when a default value cannot be provided.

```
let x;
let cond = true;
let i = 0;
loop {
    (x, cond) = foo(i);
    if (!cond) break;
    i = i + 1;
}
```

Variables must be assigned before use

Move's type system prevents a local variable from being used before it has been assigned.

```
let x;
x + x // ERROR!
```

```
let x;
if (cond) x = 0;
x + x // ERROR!
```

```
let x;
while (cond) x = 0;
x + x // ERROR!
```

Valid variable names

Variable names can contain underscores `_`, letters `a` to `z`, letters `A` to `Z`, and digits `0` to `9`. Variable names must start with either an underscore `_` or a letter `a` through `z`. They *cannot* start

with uppercase letters.

```
// all valid
let x = e;
let _x = e;
let _A = e;
let x0 = e;
let xA = e;
let foobar_123 = e;

// all invalid
let X = e; // ERROR!
let Foo = e; // ERROR!
```

Type annotations

The type of a local variable can almost always be inferred by Move's type system. However, Move allows explicit type annotations that can be useful for readability, clarity, or debuggability. The syntax for adding a type annotation is:

```
let x: T = e; // "Variable x of type T is initialized to expression e"
```

Some examples of explicit type annotations:

```
address 0x42 {
module example {

    struct S { f: u64, g: u64 }

    fun annotated() {
        let u: u8 = 0;
        let b: vector<u8> = b"hello";
        let a: address = @0x0;
        let (x, y): (&u64, &mut u64) = (&0, &mut 1);
        let S { f, g: f2 }: S = S { f: 0, g: 1 };
    }
}
}
```

Note that the type annotations must always be to the right of the pattern:

```
let (x: &u64, y: &mut u64) = (&0, &mut 1); // ERROR! should be let (x, y): ... =
```

When annotations are necessary

In some cases, a local type annotation is required if the type system cannot infer the type. This commonly occurs when the type argument for a generic type cannot be inferred. For example:

```
let _v1 = vector::empty(); // ERROR!
//        ^^^^^^^^^^^^^^^ Could not infer this type. Try adding an annotation
let v2: vector<u64> = vector::empty(); // no error
```

In a rarer case, the type system might not be able to infer a type for divergent code (where all the following code is unreachable). Both `return` and `abort` are expressions and can have any type. A `loop` has type `()` if it has a `break`, but if there is no break out of the `loop`, it could have any type. If these types cannot be inferred, a type annotation is required. For example, this code:

```
let a: u8 = return ();
let b: bool = abort 0;
let c: signer = loop ();

let x = return (); // ERROR!
//  ^ Could not infer this type. Try adding an annotation
let y = abort 0; // ERROR!
//  ^ Could not infer this type. Try adding an annotation
let z = loop (); // ERROR!
//  ^ Could not infer this type. Try adding an annotation
```

Adding type annotations to this code will expose other errors about dead code or unused local variables, but the example is still helpful for understanding this problem.

Multiple declarations with tuples

`let` can introduce more than one local at a time using tuples. The locals declared inside the parenthesis are initialized to the corresponding values from the tuple.

```
let () = ();
let (x0, x1) = (0, 1);
let (y0, y1, y2) = (0, 1, 2);
let (z0, z1, z2, z3) = (0, 1, 2, 3);
```

The type of the expression must match the arity of the tuple pattern exactly.

```
let (x, y) = (0, 1, 2); // ERROR!
let (x, y, z, q) = (0, 1, 2); // ERROR!
```

You cannot declare more than one local with the same name in a single `let`.

```
let (x, x) = 0; // ERROR!
```

Multiple declarations with structs

`let` can also introduce more than one local at a time when destructuring (or matching against) a struct. In this form, the `let` creates a set of local variables that are initialized to the values of the fields from a struct. The syntax looks like this:

```
struct T { f1: u64, f2: u64 }
```

```
let T { f1: local1, f2: local2 } = T { f1: 1, f2: 2 };
// local1: u64
// local2: u64
```

Here is a more complicated example:

```
address 0x42 {
module example {
    struct X { f: u64 }
    struct Y { x1: X, x2: X }

    fun new_x(): X {
        X { f: 1 }
    }

    fun example() {
        let Y { x1: X { f }, x2 } = Y { x1: new_x(), x2: new_x() };
        assert!(f + x2.f == 2, 42);

        let Y { x1: X { f: f1 }, x2: X { f: f2 } } = Y { x1: new_x(), x2: new_x() };
        assert!(f1 + f2 == 2, 42);
    }
}
}
```

Fields of structs can serve double duty, identifying the field to bind *and* the name of the variable. This is sometimes referred to as punning.

```
let X { f } = e;
```

is equivalent to:

```
let X { f: f } = e;
```

As shown with tuples, you cannot declare more than one local with the same name in a single `let`.

```
let Y { x1: x, x2: x } = e; // ERROR!
```

Destructuring against references

In the examples above for structs, the bound value in the let was moved, destroying the struct value and binding its fields.

```
struct T { f1: u64, f2: u64 }
```

```
let T { f1: local1, f2: local2 } = T { f1: 1, f2: 2 };
// local1: u64
// local2: u64
```

In this scenario the struct value `T { f1: 1, f2: 2 }` no longer exists after the `let`.

If you wish instead to not move and destroy the struct value, you can borrow each of its fields. For example:

```
let t = T { f1: 1, f2: 2 };
let T { f1: local1, f2: local2 } = &t;
// local1: &u64
// local2: &u64
```

And similarly with mutable references:

```
let t = T { f1: 1, f2: 2 };
let T { f1: local1, f2: local2 } = &mut t;
// local1: &mut u64
// local2: &mut u64
```

This behavior can also work with nested structs.

```
address 0x42 {
module example {
    struct X { f: u64 }
    struct Y { x1: X, x2: X }

    fun new_x(): X {
        X { f: 1 }
    }

    fun example() {
        let y = Y { x1: new_x(), x2: new_x() };

        let Y { x1: X { f }, x2 } = &y;
        assert!(*f + x2.f == 2, 42);

        let Y { x1: X { f: f1 }, x2: X { f: f2 } } = &mut y;
        *f1 = *f1 + 1;
        *f2 = *f2 + 1;
        assert!(*f1 + *f2 == 4, 42);
    }
}
}
```

Ignoring Values

In `let` bindings, it is often helpful to ignore some values. Local variables that start with `_` will be ignored and not introduce a new variable

```
fun three(): (u64, u64, u64) {
    (0, 1, 2)
}
```

```
let (x1, _, z1) = three();
let (x2, _y, z2) = three();
assert!(x1 + z1 == x2 + z2, 42);
```

This can be necessary at times as the compiler will error on unused local variables

```
let (x1, y, z1) = three(); // ERROR!
//       ^ unused local 'y'
```

General `let` grammar

All of the different structures in `let` can be combined! With that we arrive at this general grammar for `let` statements:

> *let-binding* → **let** *pattern-or-list type-annotation*$_{opt}$ *initializer*$_{opt}$ > *pattern-or-list* → *pattern* | **(** *pattern-list* **)** > *pattern-list* → *pattern* **,**$_{opt}$ | *pattern* **,** *pattern-list* > *type-annotation* → **:** *type initializer* → **=** *expression*

The general term for the item that introduces the bindings is a *pattern*. The pattern serves to both destructure data (possibly recursively) and introduce the bindings. The pattern grammar is as

follows:

> *pattern* → *local-variable* | *struct-type* **{** *field-binding-list* **}** > *field-binding-list* → *field-binding* **,**$_{opt}$ | *field-binding* **,** *field-binding-list* > *field-binding* → *field* | *field* **:** *pattern*

A few concrete examples with this grammar applied:

```
    let (x, y): (u64, u64) = (0, 1);
//       ^                            local-variable
//       ^                            pattern
//          ^                         local-variable
//          ^                         pattern
//          ^                         pattern-list
//       ^^^^                         pattern-list
//      ^^^^^^                        pattern-or-list
//            ^^^^^^^^^^^^            type-annotation
//                          ^^^^^^^^  initializer
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ let-binding

    let Foo { f, g: x } = Foo { f: 0, g: 1 };
//      ^^^                                      struct-type
//            ^                                  field
//            ^                                  field-binding
//               ^                               field
//                  ^                            local-variable
//                  ^                            pattern
//               ^^^^                            field-binding
//            ^^^^^^^                            field-binding-list
//      ^^^^^^^^^^^^^^^                          pattern
//      ^^^^^^^^^^^^^^^                          pattern-or-list
//                     ^^^^^^^^^^^^^^^^^^^^      initializer
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ let-binding
```

Mutations

Assignments

After the local is introduced (either by `let` or as a function parameter), the local can be modified via an assignment:

```
x = e
```

Unlike `let` bindings, assignments are expressions. In some languages, assignments return the value that was assigned, but in Move, the type of any assignment is always `()`.

```
(x = e: ())
```

Practically, assignments being expressions means that they can be used without adding a new expression block with braces (`{` ... `}`).

```
let x = 0;
if (cond) x = 1 else x = 2;
```

The assignment uses the same pattern syntax scheme as `let` bindings:

```
address 0x42 {
module example {
    struct X { f: u64 }

    fun new_x(): X {
        X { f: 1 }
    }

    // This example will complain about unused variables and assignments.
    fun example() {
       let (x, _, z) = (0, 1, 3);
       let (x, y, f, g);

       (X { f }, X { f: x }) = (new_x(), new_x());
       assert!(f + x == 2, 42);

       (x, y, z, f, _, g) = (0, 0, 0, 0, 0, 0);
    }
}
}
```

Note that a local variable can only have one type, so the type of the local cannot change between assignments.

```
let x;
x = 0;
x = false; // ERROR!
```

Mutating through a reference

In addition to directly modifying a local with assignment, a local can be modified via a mutable reference `&mut`.

```
let x = 0;
let r = &mut x;
*r = 1;
assert!(x == 1, 42);
```

This is particularly useful if either:

(1) You want to modify different variables depending on some condition.

```
let x = 0;
let y = 1;
let r = if (cond) &mut x else &mut y;
*r = *r + 1;
```

(2) You want another function to modify your local value.

```
let x = 0;
modify_ref(&mut x);
```

This sort of modification is how you modify structs and vectors!

```
let v = vector::empty();
vector::push_back(&mut v, 100);
assert!(*vector::borrow(&v, 0) == 100, 42);
```

For more details, see Move references.

Scopes

Any local declared with `let` is available for any subsequent expression, *within that scope*. Scopes are declared with expression blocks, `{ ... }`.

Locals cannot be used outside of the declared scope.

```
let x = 0;
{
    let y = 1;
};
x + y // ERROR!
//  ^ unbound local 'y'
```

But, locals from an outer scope *can* be used in a nested scope.

```
{
    let x = 0;
    {
        let y = x + 1; // valid
    }
}
```

Locals can be mutated in any scope where they are accessible. That mutation survives with the local, regardless of the scope that performed the mutation.

```
let x = 0;
x = x + 1;
assert!(x == 1, 42);
{
    x = x + 1;
    assert!(x == 2, 42);
};
assert!(x == 2, 42);
```

Expression Blocks

An expression block is a series of statements separated by semicolons (`;`). The resulting value of an expression block is the value of the last expression in the block.

```
{ let x = 1; let y = 1; x + y }
```

In this example, the result of the block is `x + y`.

A statement can be either a `let` declaration or an expression. Remember that assignments (`x = e`) are expressions of type `()`.

```
{ let x; let y = 1; x = 1; x + y }
```

Function calls are another common expression of type `()`. Function calls that modify data are commonly used as statements.

```
{ let v = vector::empty(); vector::push_back(&mut v, 1); v }
```

This is not just limited to `()` types---any expression can be used as a statement in a sequence!

```
{
    let x = 0;
    x + 1; // value is discarded
    x + 2; // value is discarded
    b"hello"; // value is discarded
}
```

But! If the expression contains a resource (a value without the `drop` ability), you will get an error. This is because Move's type system guarantees that any value that is dropped has the `drop` ability. (Ownership must be transferred or the value must be explicitly destroyed within its declaring module.)

```
{
    let x = 0;
    Coin { value: x }; // ERROR!
//  ^^^^^^^^^^^^^^^^^ unused value without the `drop` ability
    x
}
```

If a final expression is not present in a block---that is, if there is a trailing semicolon `;`, there is an implicit unit `()` value. Similarly, if the expression block is empty, there is an implicit unit `()` value.

```
// Both are equivalent
{ x = x + 1; 1 / x; }
{ x = x + 1; 1 / x; () }
```

```
// Both are equivalent
{ }
{ () }
```

An expression block is itself an expression and can be used anyplace an expression is used. (Note: The body of a function is also an expression block, but the function body cannot be replaced by another expression.)

```
let my_vector: vector<vector<u8>> = {
    let v = vector::empty();
    vector::push_back(&mut v, b"hello");
    vector::push_back(&mut v, b"goodbye");
    v
};
```

(The type annotation is not needed in this example and only added for clarity.)

Shadowing

If a `let` introduces a local variable with a name already in scope, that previous variable can no longer be accessed for the rest of this scope. This is called *shadowing*.

```
let x = 0;
assert!(x == 0, 42);

let x = 1; // x is shadowed
assert!(x == 1, 42);
```

When a local is shadowed, it does not need to retain the same type as before.

```
let x = 0;
assert!(x == 0, 42);

let x = b"hello"; // x is shadowed
assert!(x == b"hello", 42);
```

After a local is shadowed, the value stored in the local still exists, but will no longer be accessible. This is important to keep in mind with values of types without the `drop` ability, as ownership of the value must be transferred by the end of the function.

```
address 0x42 {
    module example {
        struct Coin has store { value: u64 }

        fun unused_resource(): Coin {
            let x = Coin { value: 0 }; // ERROR!
//              ^ This local still contains a value without the `drop` ability
            x.value = 1;
            let x = Coin { value: 10 };
            x
//          ^ Invalid return
        }
    }
}
```

When a local is shadowed inside a scope, the shadowing only remains for that scope. The shadowing is gone once that scope ends.

```
let x = 0;
{
    let x = 1;
    assert!(x == 1, 42);
};
assert!(x == 0, 42);
```

Remember, locals can change type when they are shadowed.

```
let x = 0;
{
    let x = b"hello";
    assert!(x = b"hello", 42);
};
assert!(x == 0, 42);
```

Move and Copy

All local variables in Move can be used in two ways, either by `move` or `copy`. If one or the other is not specified, the Move compiler is able to infer whether a `copy` or a `move` should be used. This means that in all of the examples above, a `move` or a `copy` would be inserted by the compiler. A local variable cannot be used without the use of `move` or `copy`.

`copy` will likely feel the most familiar coming from other programming languages, as it creates a new copy of the value inside of the variable to use in that expression. With `copy`, the local variable can be used more than once.

```
let x = 0;
let y = copy x + 1;
let z = copy x + 2;
```

Any value with the `copy` ability can be copied in this way.

`move` takes the value out of the local variable *without* copying the data. After a `move` occurs, the local variable is unavailable.

```
let x = 1;
let y = move x + 1;
//      ------ Local was moved here
let z = move x + 2; // Error!
//      ^^^^^^ Invalid usage of local 'x'
y + z
```

Safety

Move's type system will prevent a value from being used after it is moved. This is the same safety check described in `let` declaration that prevents local variables from being used before it is

assigned a value.

Inference

As mentioned above, the Move compiler will infer a `copy` or `move` if one is not indicated. The algorithm for doing so is quite simple:

- Any scalar value with the `copy` ability is given a `copy`.
- Any reference (both mutable `&mut` and immutable `&`) is given a `copy`.
    - Except under special circumstances where it is made a `move` for predictable borrow checker errors.
- Any other value is given a `move`.
    - This means that even though other values might be have the `copy` ability, it must be done *explicitly* by the programmer.
    - This is to prevent accidental copies of large data structures.

For example:

```
let s = b"hello";
let foo = Foo { f: 0 };
let coin = Coin { value: 0 };

let s2 = s; // move
let foo2 = foo; // move
let coin2 = coin; // move

let x = 0;
let b = false;
let addr = @0x42;
let x_ref = &x;
let coin_ref = &mut coin2;

let x2 = x; // copy
let b2 = b; // copy
let addr2 = @0x42; // copy
let x_ref2 = x_ref; // copy
let coin_ref2 = coin_ref; // copy
```

Equality

Move supports two equality operations `==` and `!=`

Operations

Syntax	Operation	Description
`==`	equal	Returns `true` if the two operands have the same value, `false` otherwise
`!=`	not equal	Returns `true` if the two operands have different values, `false` otherwise

Typing

Both the equal (`==`) and not-equal (`!=`) operations only work if both operands are the same type

```
0 == 0; // `true`
1u128 == 2u128; // `false`
b"hello" != x"00"; // `true`
```

Equality and non-equality also work over user defined types!

```
address 0x42 {
module example {
    struct S has copy, drop { f: u64, s: vector<u8> }

    fun always_true(): bool {
        let s = S { f: 0, s: b"" };
        // parens are not needed but added for clarity in this example
        (copy s) == s
    }

    fun always_false(): bool {
        let s = S { f: 0, s: b"" };
        // parens are not needed but added for clarity in this example
        (copy s) != s
    }
}
}
```

If the operands have different types, there is a type checking error

```
1u8 == 1u128; // ERROR!
//     ^^^^^ expected an argument of type 'u8'
b"" != 0; // ERROR!
//     ^ expected an argument of type 'vector<u8>'
```

Typing with references

When comparing references, the type of the reference (immutable or mutable) does not matter. This means that you can compare an immutable `&` reference with a mutable one `&mut` of the same underlying type.

```
let i = &0;
let m = &mut 1;

i == m; // `false`
m == i; // `false`
m == m; // `true`
i == i; // `true`
```

The above is equivalent to applying an explicit freeze to each mutable reference where needed

```
let i = &0;
let m = &mut 1;

i == freeze(m); // `false`
freeze(m) == i; // `false`
m == m; // `true`
i == i; // `true`
```

But again, the underlying type must be the same type

```
let i = &0;
let s = &b"";

i == s; // ERROR!
//   ^ expected an argument of type '&u64'
```

Restrictions

Both `==` and `!=` consume the value when comparing them. As a result, the type system enforces that the type must have `drop`. Recall that without the `drop` ability, ownership must be transferred by the end of the function, and such values can only be explicitly destroyed within their declaring

module. If these were used directly with either equality `==` or non-equality `!=`, the value would be destroyed which would break `drop` ability safety guarantees!

```
address 0x42 {
module example {
    struct Coin has store { value: u64 }
    fun invalid(c1: Coin, c2: Coin) {
        c1 == c2 // ERROR!
//      ^^    ^^ These resources would be destroyed!
    }
}
}
```

But, a programmer can *always* borrow the value first instead of directly comparing the value, and reference types have the `drop` ability. For example

```
address 0x42 {
module example {
    struct Coin as store { value: u64 }
    fun swap_if_equal(c1: Coin, c2: Coin): (Coin, Coin) {
        let are_equal = &c1 == &c2; // valid
        if (are_equal) (c2, c1) else (c1, c2)
    }
}
}
```

Avoid Extra Copies

While a programmer *can* compare any value whose type has `drop`, a programmer should often compare by reference to avoid expensive copies.

```
let v1: vector<u8> = function_that_returns_vector();
let v2: vector<u8> = function_that_returns_vector();
assert!(copy v1 == copy v2, 42);
//     ^^^^       ^^^^
use_two_vectors(v1, v2);

let s1: Foo = function_that_returns_large_struct();
let s2: Foo = function_that_returns_large_struct();
assert!(copy s1 == copy s2, 42);
//     ^^^^       ^^^^
use_two_foos(s1, s2);
```

This code is perfectly acceptable (assuming `Foo` has `drop`), just not efficient. The highlighted copies can be removed and replaced with borrows

```
let v1: vector<u8> = function_that_returns_vector();
let v2: vector<u8> = function_that_returns_vector();
assert!(&v1 == &v2, 42);
//     ^      ^
use_two_vectors(v1, v2);

let s1: Foo = function_that_returns_large_struct();
let s2: Foo = function_that_returns_large_struct();
assert!(&s1 == &s2, 42);
//     ^      ^
use_two_foos(s1, s2);
```

The efficiency of the `==` itself remains the same, but the `copy`s are removed and thus the program is more efficient.

Abort and Assert

`return` and `abort` are two control flow constructs that end execution, one for the current function and one for the entire transaction.

More information on `return` can be found in the linked section

abort

`abort` is an expression that takes one argument: an **abort code** of type `u64`. For example:

```
abort 42
```

The `abort` expression halts execution the current function and reverts all changes made to global state by the current transaction. There is no mechanism for "catching" or otherwise handling an `abort`.

Luckily, in Move transactions are all or nothing, meaning any changes to global storage are made all at once only if the transaction succeeds. Because of this transactional commitment of changes, after an abort there is no need to worry about backing out changes. While this approach is lacking in flexibility, it is incredibly simple and predictable.

Similar to `return`, `abort` is useful for exiting control flow when some condition cannot be met.

In this example, the function will pop two items off of the vector, but will abort early if the vector does not have two items

```
use std::vector;
fun pop_twice<T>(v: &mut vector<T>): (T, T) {
    if (vector::length(v) < 2) abort 42;

    (vector::pop_back(v), vector::pop_back(v))
}
```

This is even more useful deep inside a control-flow construct. For example, this function checks that all numbers in the vector are less than the specified `bound`. And aborts otherwise

```
use std::vector;
fun check_vec(v: &vector<u64>, bound: u64) {
    let i = 0;
    let n = vector::length(v);
    while (i < n) {
        let cur = *vector::borrow(v, i);
        if (cur > bound) abort 42;
        i = i + 1;
    }
}
```

assert

`assert` is a builtin, macro-like operation provided by the Move compiler. It takes two arguments, a condition of type `bool` and a code of type `u64`

```
assert!(condition: bool, code: u64)
```

Since the operation is a macro, it must be invoked with the `!`. This is to convey that the arguments to `assert` are call-by-expression. In other words, `assert` is not a normal function and does not exist at the bytecode level. It is replaced inside the compiler with

```
if (condition) () else abort code
```

`assert` is more commonly used than just `abort` by itself. The `abort` examples above can be rewritten using `assert`

```
use std::vector;
fun pop_twice<T>(v: &mut vector<T>): (T, T) {
    assert!(vector::length(v) >= 2, 42); // Now uses 'assert'

    (vector::pop_back(v), vector::pop_back(v))
}
```

and

```
use std::vector;
fun check_vec(v: &vector<u64>, bound: u64) {
    let i = 0;
    let n = vector::length(v);
    while (i < n) {
        let cur = *vector::borrow(v, i);
        assert!(cur <= bound, 42); // Now uses 'assert'
        i = i + 1;
    }
}
```

Note that because the operation is replaced with this `if-else`, the argument for the `code` is not always evaluated. For example:

```
assert!(true, 1 / 0)
```

Will not result in an arithmetic error, it is equivalent to

```
if (true) () else (1 / 0)
```

So the arithmetic expression is never evaluated!

Abort codes in the Move VM

When using `abort`, it is important to understand how the `u64` code will be used by the VM.

Normally, after successful execution, the Move VM produces a change-set for the changes made to global storage (added/removed resources, updates to existing resources, etc).

If an `abort` is reached, the VM will instead indicate an error. Included in that error will be two pieces of information:

- The module that produced the abort (address and name)
- The abort code.

For example

```
address 0x2 {
module example {
    public fun aborts() {
        abort 42
    }
}
}

script {
    fun always_aborts() {
        0x2::example::aborts()
    }
}
```

If a transaction, such as the script `always_aborts` above, calls `0x2::example::aborts`, the VM would produce an error that indicated the module `0x2::example` and the code `42`.

This can be useful for having multiple aborts being grouped together inside a module.

In this example, the module has two separate error codes used in multiple functions

```
address 0x42 {
module example {

    use std::vector;

    const EMPTY_VECTOR: u64 = 0;
    const INDEX_OUT_OF_BOUNDS: u64 = 1;

    // move i to j, move j to k, move k to i
    public fun rotate_three<T>(v: &mut vector<T>, i: u64, j: u64, k: u64) {
        let n = vector::length(v);
        assert!(n > 0, EMPTY_VECTOR);
        assert!(i < n, INDEX_OUT_OF_BOUNDS);
        assert!(j < n, INDEX_OUT_OF_BOUNDS);
        assert!(k < n, INDEX_OUT_OF_BOUNDS);

        vector::swap(v, i, k);
        vector::swap(v, j, k);
    }

    public fun remove_twice<T>(v: &mut vector<T>, i: u64, j: u64): (T, T) {
        let n = vector::length(v);
        assert!(n > 0, EMPTY_VECTOR);
        assert!(i < n, INDEX_OUT_OF_BOUNDS);
        assert!(j < n, INDEX_OUT_OF_BOUNDS);
        assert!(i > j, INDEX_OUT_OF_BOUNDS);

        (vector::remove<T>(v, i), vector::remove<T>(v, j))
    }
}
}
```

The type of `abort`

The `abort i` expression can have any type! This is because both constructs break from the normal control flow, so they never need to evaluate to the value of that type.

The following are not useful, but they will type check

```
let y: address = abort 0;
```

This behavior can be helpful in situations where you have a branching instruction that produces a value on some branches, but not all. For example:

```
let b =
    if (x == 0) false
    else if (x == 1) true
    else abort 42;
//       ^^^^^^^^ `abort 42` has type `bool`
```

Conditionals

An `if` expression specifies that some code should only be evaluated if a certain condition is true. For example:

```
if (x > 5) x = x - 5
```

The condition must be an expression of type `bool`.

An `if` expression can optionally include an `else` clause to specify another expression to evaluate when the condition is false.

```
if (y <= 10) y = y + 1 else y = 10
```

Either the "true" branch or the "false" branch will be evaluated, but not both. Either branch can be a single expression or an expression block.

The conditional expressions may produce values so that the `if` expression has a result.

```
let z = if (x < 100) x else 100;
```

The expressions in the true and false branches must have compatible types. For example:

```
// x and y must be u64 integers
let maximum: u64 = if (x > y) x else y;

// ERROR! branches different types
let z = if (maximum < 10) 10u8 else 100u64;

// ERROR! branches different types, as default false-branch is () not u64
if (maximum >= 10) maximum;
```

If the `else` clause is not specified, the false branch defaults to the unit value. The following are equivalent:

```
if (condition) true_branch // implied default: else ()
if (condition) true_branch else ()
```

Commonly, `if` expressions are used in conjunction with expression blocks.

```
let maximum = if (x > y) x else y;
if (maximum < 10) {
    x = x + 10;
    y = y + 10;
} else if (x >= 10 && y >= 10) {
    x = x - 10;
    y = y - 10;
}
```

Grammar for Conditionals

if-expression → **if (** *expression* **)** *expression else-clause*$_{opt}$ *else-clause* → **else** *expression*

While and Loop

Move offers two constructs for looping: `while` and `loop`.

`while` loops

The `while` construct repeats the body (an expression of type unit) until the condition (an expression of type `bool`) evaluates to `false`.

Here is an example of simple `while` loop that computes the sum of the numbers from `1` to `n`:

```
fun sum(n: u64): u64 {
    let sum = 0;
    let i = 1;
    while (i <= n) {
        sum = sum + i;
        i = i + 1
    };

    sum
}
```

Infinite loops are allowed:

```
fun foo() {
    while (true) { }
}
```

break

The `break` expression can be used to exit a loop before the condition evaluates to `false`. For example, this loop uses `break` to find the smallest factor of `n` that's greater than 1:

```
fun smallest_factor(n: u64): u64 {
    // assuming the input is not 0 or 1
    let i = 2;
    while (i <= n) {
        if (n % i == 0) break;
        i = i + 1
    };

    i
}
```

The `break` expression cannot be used outside of a loop.

continue

The `continue` expression skips the rest of the loop and continues to the next iteration. This loop uses `continue` to compute the sum of `1, 2, ..., n`, except when the number is divisible by 10:

```
fun sum_intermediate(n: u64): u64 {
    let sum = 0;
    let i = 0;
    while (i < n) {
        i = i + 1;
        if (i % 10 == 0) continue;
        sum = sum + i;
    };

    sum
}
```

The `continue` expression cannot be used outside of a loop.

The type of `break` and `continue`

`break` and `continue`, much like `return` and `abort`, can have any type. The following examples illustrate where this flexible typing can be helpful:

```
fun pop_smallest_while_not_equal(
    v1: vector<u64>,
    v2: vector<u64>,
): vector<u64> {
    let result = vector::empty();
    while (!vector::is_empty(&v1) && !vector::is_empty(&v2)) {
        let u1 = *vector::borrow(&v1, vector::length(&v1) - 1);
        let u2 = *vector::borrow(&v2, vector::length(&v2) - 1);
        let popped =
            if (u1 < u2) vector::pop_back(&mut v1)
            else if (u2 < u1) vector::pop_back(&mut v2)
            else break; // Here, `break` has type `u64`
        vector::push_back(&mut result, popped);
    };

    result
}
```

```
fun pick(
    indexes: vector<u64>,
    v1: &vector<address>,
    v2: &vector<address>
): vector<address> {
    let len1 = vector::length(v1);
    let len2 = vector::length(v2);
    let result = vector::empty();
    while (!vector::is_empty(&indexes)) {
        let index = vector::pop_back(&mut indexes);
        let chosen_vector =
            if (index < len1) v1
            else if (index < len2) v2
            else continue; // Here, `continue` has type `&vector<address>`
        vector::push_back(&mut result, *vector::borrow(chosen_vector, index))
    };

    result
}
```

The `loop` expression

The `loop` expression repeats the loop body (an expression with type `()`) until it hits a `break`

Without a `break`, the loop will continue forever

```
fun foo() {
    let i = 0;
    loop { i = i + 1 }
}
```

Here is an example that uses `loop` to write the `sum` function:

```
fun sum(n: u64): u64 {
    let sum = 0;
    let i = 0;
    loop {
        i = i + 1;
        if (i > n) break;
        sum = sum + i
    };

    sum
}
```

As you might expect, `continue` can also be used inside a `loop`. Here is `sum_intermediate` from above rewritten using `loop` instead of `while`

```
fun sum_intermediate(n: u64): u64 {
    let sum = 0;
    let i = 0;
    loop {
        i = i + 1;
        if (i % 10 == 0) continue;
        if (i > n) break;
        sum = sum + i
    };

    sum
}
```

The type of `while` and `loop`

Move loops are typed expressions. A `while` expression always has type `()`.

```
let () = while (i < 10) { i = i + 1 };
```

If a `loop` contains a `break`, the expression has type unit `()`

```
(loop { if (i < 10) i = i + 1 else break }: ());
let () = loop { if (i < 10) i = i + 1 else break };
```

If `loop` does not have a `break`, `loop` can have any type much like `return`, `abort`, `break`, and `continue`.

```
(loop (): u64);
(loop (): address);
(loop (): &vector<vector<u8>>);
```

Functions

Function syntax in Move is shared between module functions and script functions. Functions inside of modules are reusable, whereas script functions are only used once to invoke a transaction.

Declaration

Functions are declared with the `fun` keyword followed by the function name, type parameters, parameters, a return type, acquires annotations, and finally the function body.

```
fun <identifier><[type_parameters: constraint],*>([identifier: type],*): <return_type>
<acquires [identifier],*> <function_body>
```

For example

```
fun foo<T1, T2>(x: u64, y: T1, z: T2): (T2, T1, u64) { (z, y, x) }
```

Visibility

Module functions, by default, can only be called within the same module. These internal (sometimes called private) functions cannot be called from other modules or from scripts.

```
address 0x42 {
module m {
    fun foo(): u64 { 0 }
    fun calls_foo(): u64 { foo() } // valid
}

module other {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // ERROR!
//      ^^^^^^^^^^^^ 'foo' is internal to '0x42::m'
    }
}
}

script {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // ERROR!
//      ^^^^^^^^^^^^ 'foo' is internal to '0x42::m'
    }
}
```

To allow access from other modules or from scripts, the function must be declared `public` or `public(friend)`.

`public` visibility

A `public` function can be called by *any* function defined in *any* module or script. As shown in the following example, a `public` function can be called by:

- other functions defined in the same module,
- functions defined in another module, or
- the function defined in a script.

```
address 0x42 {
module m {
    public fun foo(): u64 { 0 }
    fun calls_foo(): u64 { foo() } // valid
}

module other {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // valid
    }
}
}

script {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // valid
    }
}
```

`public(friend)` visibility

The `public(friend)` visibility modifier is a more restricted form of the `public` modifier to give more control about where a function can be used. A `public(friend)` function can be called by:

- other functions defined in the same module, or
- functions defined in modules which are explicitly specified in the **friend list** (see Friends on how to specify the friend list).

Note that since we cannot declare a script to be a friend of a module, the functions defined in scripts can never call a `public(friend)` function.

```
address 0x42 {
module m {
    friend 0x42::n;  // friend declaration
    public(friend) fun foo(): u64 { 0 }
    fun calls_foo(): u64 { foo() } // valid
}

module n {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // valid
    }
}

module other {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // ERROR!
//      ^^^^^^^^^^^^ 'foo' can only be called from a 'friend' of module '0x42::m'
    }
}
}

script {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // ERROR!
//      ^^^^^^^^^^^^ 'foo' can only be called from a 'friend' of module '0x42::m'
    }
}
```

`entry` modifier

The `entry` modifier is designed to allow module functions to be safely and directly invoked much like scripts. This allows module writers to specify which functions can be to begin execution. The module writer then knows that any non-`entry` function will be called from a Move program already in execution.

Essentially, `entry` functions are the "main" functions of a module, and they specify where Move programs start executing.

Note though, an `entry` function *can* still be called by other Move functions. So while they *can* serve as the start of a Move program, they aren't restricted to that case.

For example:

```
address 0x42 {
module m {
    public entry fun foo(): u64 { 0 }
    fun calls_foo(): u64 { foo() } // valid!
}

module n {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // valid!
    }
}

module other {
    public entry fun calls_m_foo(): u64 {
        0x42::m::foo() // valid!
    }
}
}

script {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // valid!
    }
}
```

Even internal functions can be marked as `entry`! This lets you guarantee that the function is called only at the beginning of execution (assuming you do not call it elsewhere in your module)

```
address 0x42 {
module m {
    entry fun foo(): u64 { 0 } // valid! entry functions do not have to be public
}

module n {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // ERROR!
//      ^^^^^^^^^^^^ 'foo' is internal to '0x42::m'
    }
}

module other {
    public entry fun calls_m_foo(): u64 {
        0x42::m::foo() // ERROR!
//      ^^^^^^^^^^^^ 'foo' is internal to '0x42::m'
    }
}
}

script {
    fun calls_m_foo(): u64 {
        0x42::m::foo() // ERROR!
//      ^^^^^^^^^^^^ 'foo' is internal to '0x42::m'
    }
}
```

Name

Function names can start with letters `a` to `z` or letters `A` to `Z`. After the first character, function names can contain underscores `_`, letters `a` to `z`, letters `A` to `Z`, or digits `0` to `9`.

```
fun FOO() {}
fun bar_42() {}
fun _bAZ19() {}
    ^^^^^^ Invalid function name '_bAZ19'. Function names cannot start with '_'
```

Type Parameters

After the name, functions can have type parameters

```
fun id<T>(x: T): T { x }
fun example<T1: copy, T2>(x: T1, y: T2): (T1, T1, T2) { (copy x, x, y) }
```

For more details, see Move generics.

Parameters

Functions parameters are declared with a local variable name followed by a type annotation

```
fun add(x: u64, y: u64): u64 { x + y }
```

We read this as `x` has type `u64`

A function does not have to have any parameters at all.

```
fun useless() { }
```

This is very common for functions that create new or empty data structures

```
address 0x42 {
module example {
  struct Counter { count: u64 }

  fun new_counter(): Counter {
      Counter { count: 0 }
  }

}
}
```

Acquires

When a function accesses a resource using `move_from`, `borrow_global`, or `borrow_global_mut`, the function must indicate that it `acquires` that resource. This is then used by Move's type system to ensure the references into global storage are safe, specifically that there are no dangling references into global storage.

```
address 0x42 {
module example {

    struct Balance has key { value: u64 }

    public fun add_balance(s: &signer, value: u64) {
        move_to(s, Balance { value })
    }

    public fun extract_balance(addr: address): u64 acquires Balance {
        let Balance { value } = move_from(addr); // acquires needed
        value
    }
}
}
```

`acquires` annotations must also be added for transitive calls within the module. Calls to these functions from another module do not need to annotated with these acquires because one module cannot access resources declared in another module--so the annotation is not needed to ensure reference safety.

```
address 0x42 {
module example {

    struct Balance has key { value: u64 }

    public fun add_balance(s: &signer, value: u64) {
        move_to(s, Balance { value })
    }

    public fun extract_balance(addr: address): u64 acquires Balance {
        let Balance { value } = move_from(addr); // acquires needed
        value
    }

    public fun extract_and_add(sender: address, receiver: &signer) acquires Balance {
        let value = extract_balance(sender); // acquires needed here
        add_balance(receiver, value)
    }
}
}

address 0x42 {
module other {
    fun extract_balance(addr: address): u64 {
        0x42::example::extract_balance(addr) // no acquires needed
    }
}
}
```

A function can `acquire` as many resources as it needs to

```
address 0x42 {
module example {
    use std::vector;

    struct Balance has key { value: u64 }
    struct Box<T> has key { items: vector<T> }

    public fun store_two<Item1: store, Item2: store>(
        addr: address,
        item1: Item1,
        item2: Item2,
    ) acquires Balance, Box {
        let balance = borrow_global_mut<Balance>(addr); // acquires needed
        balance.value = balance.value - 2;
        let box1 = borrow_global_mut<Box<Item1>>(addr); // acquires needed
        vector::push_back(&mut box1.items, item1);
        let box2 = borrow_global_mut<Box<Item2>>(addr); // acquires needed
        vector::push_back(&mut box2.items, item2);
    }
}
}
```

Return type

After the parameters, a function specifies its return type.

```
fun zero(): u64 { 0 }
```

Here `: u64` indicates that the function's return type is `u64`.

Using tuples, a function can return multiple values:

```
fun one_two_three(): (u64, u64, u64) { (0, 1, 2) }
```

If no return type is specified, the function has an implicit return type of unit `()`. These functions are equivalent:

```
fun just_unit(): () { () }
fun just_unit() { () }
fun just_unit() { }
```

`script` functions must have a return type of unit `()`:

```
script {
    fun do_nothing() {
    }
}
```

As mentioned in the tuples section, these tuple "values" are virtual and do not exist at runtime. So for a function that returns unit `()`, it will not be returning any value at all during execution.

Function body

A function's body is an expression block. The return value of the function is the last value in the sequence

```
fun example(): u64 {
    let x = 0;
    x = x + 1;
    x // returns 'x'
}
```

See the section below for more information on returns

For more information on expression blocks, see Move variables.

Native Functions

Some functions do not have a body specified, and instead have the body provided by the VM. These functions are marked `native`.

Without modifying the VM source code, a programmer cannot add new native functions. Furthermore, it is the intent that `native` functions are used for either standard library code or for functionality needed for the given Move environment.

Most `native` functions you will likely see are in standard library code such as `vector`

```
module std::vector {
    native public fun empty<Element>(): vector<Element>;
    ...
}
```

Calling

When calling a function, the name can be specified either through an alias or fully qualified

```
address 0x42 {
module example {
    public fun zero(): u64 { 0 }
}
}

script {
    use 0x42::example::{Self, zero};
    fun call_zero() {
        // With the `use` above all of these calls are equivalent
        0x42::example::zero();
        example::zero();
        zero();
    }
}
```

When calling a function, an argument must be given for every parameter.

```
address 0x42 {
module example {
    public fun takes_none(): u64 { 0 }
    public fun takes_one(x: u64): u64 { x }
    public fun takes_two(x: u64, y: u64): u64 { x + y }
    public fun takes_three(x: u64, y: u64, z: u64): u64 { x + y + z }
}
}

script {
    use 0x42::example;
    fun call_all() {
        example::takes_none();
        example::takes_one(0);
        example::takes_two(0, 1);
        example::takes_three(0, 1, 2);
    }
}
```

Type arguments can be either specified or inferred. Both calls are equivalent.

```
address 0x42 {
module example {
    public fun id<T>(x: T): T { x }
}
}

script {
    use 0x42::example;
    fun call_all() {
        example::id(0);
        example::id<u64>(0);
    }
}
```

For more details, see Move generics.

Returning values

The result of a function, its "return value", is the final value of its function body. For example

```
fun add(x: u64, y: u64): u64 {
    x + y
}
```

As mentioned above, the function's body is an expression block. The expression block can sequence various statements, and the final expression in the block will be be the value of that block

```
fun double_and_add(x: u64, y: u64): u64 {
    let double_x = x * 2;
    let double_y = y * 2;
    double_x + double_y
}
```

The return value here is `double_x + double_y`

`return` expression

A function implicitly returns the value that its body evaluates to. However, functions can also use the explicit `return` expression:

```
fun f1(): u64 { return 0 }
fun f2(): u64 { 0 }
```

These two functions are equivalent. In this slightly more involved example, the function subtracts two `u64` values, but returns early with `0` if the second value is too large:

```
fun safe_sub(x: u64, y: u64): u64 {
    if (y > x) return 0;
    x - y
}
```

Note that the body of this function could also have been written as `if (y > x) 0 else x - y`.

However `return` really shines is in exiting deep within other control flow constructs. In this example, the function iterates through a vector to find the index of a given value:

```
use std::vector;
use std::option::{Self, Option};
fun index_of<T>(v: &vector<T>, target: &T): Option<u64> {
    let i = 0;
    let n = vector::length(v);
    while (i < n) {
        if (vector::borrow(v, i) == target) return option::some(i);
        i = i + 1
    };

    option::none()
}
```

Using `return` without an argument is shorthand for `return ()`. That is, the following two functions are equivalent:

```
fun foo() { return }
fun foo() { return () }
```

Structs and Resources

A *struct* is a user-defined data structure containing typed fields. Structs can store any non-reference type, including other structs.

We often refer to struct values as *resources* if they cannot be copied and cannot be dropped. In this case, resource values must have ownership transferred by the end of the function. This property makes resources particularly well served for defining global storage schemas or for representing important values (such as a token).

By default, structs are linear and ephemeral. By this we mean that they: cannot be copied, cannot be dropped, and cannot be stored in global storage. This means that all values have to have ownership transferred (linear) and the values must be dealt with by the end of the program's execution (ephemeral). We can relax this behavior by giving the struct abilities which allow values to be copied or dropped and also to be stored in global storage or to define global storage schemas.

Defining Structs

Structs must be defined inside a module:

```
address 0x2 {
module m {
    struct Foo { x: u64, y: bool }
    struct Bar {}
    struct Baz { foo: Foo, }
    //                   ^ note: it is fine to have a trailing comma
}
}
```

Structs cannot be recursive, so the following definition is invalid:

```
struct Foo { x: Foo }
//              ^ error! Foo cannot contain Foo
```

As mentioned above: by default, a struct declaration is linear and ephemeral. So to allow the value to be used with certain operations (that copy it, drop it, store it in global storage, or use it as a storage schema), structs can be granted abilities by annotating them with `has <ability>`:

```
address 0x2 {
module m {
    struct Foo has copy, drop { x: u64, y: bool }
}
}
```

For more details, see the annotating structs section.

Naming

Structs must start with a capital letter `A` to `Z`. After the first letter, struct names can contain underscores `_`, letters `a` to `z`, letters `A` to `Z`, or digits `0` to `9`.

```
struct Foo {}
struct BAR {}
struct B_a_z_4_2 {}
```

This naming restriction of starting with `A` to `Z` is in place to give room for future language features. It may or may not be removed later.

Using Structs

Creating Structs

Values of a struct type can be created (or "packed") by indicating the struct name, followed by value for each field:

```
address 0x2 {
module m {
    struct Foo has drop { x: u64, y: bool }
    struct Baz has drop { foo: Foo }

    fun example() {
        let foo = Foo { x: 0, y: false };
        let baz = Baz { foo: foo };
    }
}
}
```

If you initialize a struct field with a local variable whose name is the same as the field, you can use the following shorthand:

```
let baz = Baz { foo: foo };
// is equivalent to
let baz = Baz { foo };
```

This is called sometimes called "field name punning".

Destroying Structs via Pattern Matching

Struct values can be destroyed by binding or assigning them patterns.

```
address 0x2 {
module m {
    struct Foo { x: u64, y: bool }
    struct Bar { foo: Foo }
    struct Baz {}

    fun example_destroy_foo() {
        let foo = Foo { x: 3, y: false };
        let Foo { x, y: foo_y } = foo;
        //        ^ shorthand for `x: x`

        // two new bindings
        //   x: u64 = 3
        //   foo_y: bool = false
    }

    fun example_destroy_foo_wildcard() {
        let foo = Foo { x: 3, y: false };
        let Foo { x, y: _ } = foo;

        // only one new binding since y was bound to a wildcard
        //   x: u64 = 3
    }

    fun example_destroy_foo_assignment() {
        let x: u64;
        let y: bool;
        Foo { x, y } = Foo { x: 3, y: false };

        // mutating existing variables x & y
        //   x = 3, y = false
    }

    fun example_foo_ref() {
        let foo = Foo { x: 3, y: false };
        let Foo { x, y } = &foo;

        // two new bindings
        //   x: &u64
```

```
        //   y: &bool
    }

    fun example_foo_ref_mut() {
        let foo = Foo { x: 3, y: false };
        let Foo { x, y } = &mut foo;

        // two new bindings
        //   x: &mut u64
        //   y: &mut bool
    }

    fun example_destroy_bar() {
        let bar = Bar { foo: Foo { x: 3, y: false } };
        let Bar { foo: Foo { x, y } } = bar;
        //             ^ nested pattern

        // two new bindings
        //   x: u64 = 3
        //   y: bool = false
    }

    fun example_destroy_baz() {
        let baz = Baz {};
        let Baz {} = baz;
    }
}
}
```

Borrowing Structs and Fields

The `&` and `&mut` operator can be used to create references to structs or fields. These examples include some optional type annotations (e.g., `: &Foo`) to demonstrate the type of operations.

```
let foo = Foo { x: 3, y: true };
let foo_ref: &Foo = &foo;
let y: bool = foo_ref.y;          // reading a field via a reference to the struct
let x_ref: &u64 = &foo.x;

let x_ref_mut: &mut u64 = &mut foo.x;
*x_ref_mut = 42;            // modifying a field via a mutable reference
```

It is possible to borrow inner fields of nested structs:

```
let foo = Foo { x: 3, y: true };
let bar = Bar { foo };

let x_ref = &bar.foo.x;
```

You can also borrow a field via a reference to a struct:

```
let foo = Foo { x: 3, y: true };
let foo_ref = &foo;
let x_ref = &foo_ref.x;
// this has the same effect as let x_ref = &foo.x
```

Reading and Writing Fields

If you need to read and copy a field's value, you can then dereference the borrowed field:

```
let foo = Foo { x: 3, y: true };
let bar = Bar { foo: copy foo };
let x: u64 = *&foo.x;
let y: bool = *&foo.y;
let foo2: Foo = *&bar.foo;
```

If the field is implicitly copyable, the dot operator can be used to read fields of a struct without any borrowing. (Only scalar values with the `copy` ability are implicitly copyable.)

```
let foo = Foo { x: 3, y: true };
let x = foo.x;  // x == 3
let y = foo.y;  // y == true
```

Dot operators can be chained to access nested fields:

```
let baz = Baz { foo: Foo { x: 3, y: true } };
let x = baz.foo.x; // x = 3;
```

However, this is not permitted for fields that contain non-primitive types, such a vector or another struct:

```
let foo = Foo { x: 3, y: true };
let bar = Bar { foo };
let foo2: Foo = *&bar.foo;
let foo3: Foo = bar.foo; // error! must add an explicit copy with *&
```

The reason behind this design decision is that copying a vector or another struct might be an expensive operation. It is important for a programmer to be aware of this copy and make others aware with the explicit syntax `*&`.

In addition reading from fields, the dot syntax can be used to modify fields, regardless of the field being a primitive type or some other struct.

```
let foo = Foo { x: 3, y: true };
foo.x = 42;     // foo = Foo { x: 42, y: true }
foo.y = !foo.y; // foo = Foo { x: 42, y: false }
let bar = Bar { foo };            // bar = Bar { foo: Foo { x: 42, y: false } }
bar.foo.x = 52;                   // bar = Bar { foo: Foo { x: 52, y: false } }
bar.foo = Foo { x: 62, y: true }; // bar = Bar { foo: Foo { x: 62, y: true } }
```

The dot syntax also works via a reference to a struct:

```
let foo = Foo { x: 3, y: true };
let foo_ref = &mut foo;
foo_ref.x = foo_ref.x + 1;
```

Privileged Struct Operations

Most struct operations on a struct type `T` can only be performed inside the module that declares `T`:

- Struct types can only be created ("packed"), destroyed ("unpacked") inside the module that defines the struct.
- The fields of a struct are only accessible inside the module that defines the struct.

Following these rules, if you want to modify your struct outside the module, you will need to provide public APIs for them. The end of the chapter contains some examples of this.

However, struct *types* are always visible to another module or script:

```
// m.move
address 0x2 {
module m {
    struct Foo has drop { x: u64 }

    public fun new_foo(): Foo {
        Foo { x: 42 }
    }
}
}
```

```
// n.move
address 0x2 {
module n {
    use 0x2::m;

    struct Wrapper has drop {
        foo: m::Foo
    }

    fun f1(foo: m::Foo) {
        let x = foo.x;
        //      ^ error! cannot access fields of `foo` here
    }

    fun f2() {
        let foo_wrapper = Wrapper { foo: m::new_foo() };
    }
}
}
```

Note that structs do not have visibility modifiers (e.g., `public` or `private`).

Ownership

As mentioned above in Defining Structs, structs are by default linear and ephemeral. This means they cannot be copied or dropped. This property can be very useful when modeling real world resources like money, as you do not want money to be duplicated or get lost in circulation.

```
address 0x2 {
module m {
    struct Foo { x: u64 }

    public fun copying_resource() {
        let foo = Foo { x: 100 };
        let foo_copy = copy foo; // error! 'copy'-ing requires the 'copy' ability
        let foo_ref = &foo;
        let another_copy = *foo_ref // error! dereference requires the 'copy' ability
    }

    public fun destroying_resource1() {
        let foo = Foo { x: 100 };

        // error! when the function returns, foo still contains a value.
        // This destruction requires the 'drop' ability
    }

    public fun destroying_resource2(f: &mut Foo) {
        *f = Foo { x: 100 } // error!
                            // destroying the old value via a write requires the 'drop'
ability
    }
}
}
```

To fix the second example (`fun destroying_resource1`), you would need to manually "unpack" the resource:

```
address 0x2 {
module m {
    struct Foo { x: u64 }

    public fun destroying_resource1_fixed() {
        let foo = Foo { x: 100 };
        let Foo { x: _ } = foo;
    }
}
}
```

Recall that you are only able to deconstruct a resource within the module in which it is defined. This can be leveraged to enforce certain invariants in a system, for example, conservation of money.

If on the other hand, your struct does not represent something valuable, you can add the abilities `copy` and `drop` to get a struct value that might feel more familiar from other programming languages:

```
address 0x2 {
module m {
    struct Foo has copy, drop { x: u64 }

    public fun run() {
        let foo = Foo { x: 100 };
        let foo_copy = copy foo;
        // ^ this code copies foo, whereas `let x = foo` or
        // `let x = move foo` both move foo

        let x = foo.x;            // x = 100
        let x_copy = foo_copy.x;  // x = 100

        // both foo and foo_copy are implicitly discarded when the function returns
    }
}
}
```

Storing Resources in Global Storage

Only structs with the `key` ability can be saved directly in persistent global storage. All values stored within those `key` structs must have the `store` ability. See the ability and global storage chapters for more detail.

Examples

Here are two short examples of how you might use structs to represent valuable data (in the case of `Coin`) or more classical data (in the case of `Point` and `Circle`).

Example 1: Coin

```
address 0x2 {
module m {
    // We do not want the Coin to be copied because that would be duplicating this
"money",
    // so we do not give the struct the 'copy' ability.
    // Similarly, we do not want programmers to destroy coins, so we do not give the
struct the
    // 'drop' ability.
    // However, we *want* users of the modules to be able to store this coin in
persistent global
    // storage, so we grant the struct the 'store' ability. This struct will only be
inside of
    // other resources inside of global storage, so we do not give the struct the 'key'
ability.
    struct Coin has store {
        value: u64,
    }

    public fun mint(value: u64): Coin {
        // You would want to gate this function with some form of access control to
prevent
        // anyone using this module from minting an infinite amount of coins.
        Coin { value }
    }

    public fun withdraw(coin: &mut Coin, amount: u64): Coin {
        assert!(coin.balance >= amount, 1000);
        coin.value = coin.value - amount;
        Coin { value: amount }
    }

    public fun deposit(coin: &mut Coin, other: Coin) {
        let Coin { value } = other;
        coin.value = coin.value + value;
    }

    public fun split(coin: Coin, amount: u64): (Coin, Coin) {
```

```
        let other = withdraw(&mut coin, amount);
        (coin, other)
    }

    public fun merge(coin1: Coin, coin2: Coin): Coin {
        deposit(&mut coin1, coin2);
        coin1
    }

    public fun destroy_zero(coin: Coin) {
        let Coin { value } = coin;
        assert!(value == 0, 1001);
    }
}
}
```

Example 2: Geometry

```
address 0x2 {
module point {
    struct Point has copy, drop, store {
        x: u64,
        y: u64,
    }

    public fun new(x: u64, y: u64): Point {
        Point {
            x, y
        }
    }

    public fun x(p: &Point): u64 {
        p.x
    }

    public fun y(p: &Point): u64 {
        p.y
    }

    fun abs_sub(a: u64, b: u64): u64 {
        if (a < b) {
            b - a
        }
        else {
            a - b
        }
    }

    public fun dist_squared(p1: &Point, p2: &Point): u64 {
        let dx = abs_sub(p1.x, p2.x);
        let dy = abs_sub(p1.y, p2.y);
        dx*dx + dy*dy
    }
```

```
}
}
```

```
address 0x2 {
module circle {
    use 0x2::point::{Self, Point};

    struct Circle has copy, drop, store {
        center: Point,
        radius: u64,
    }

    public fun new(center: Point, radius: u64): Circle {
        Circle { center, radius }
    }

    public fun overlaps(c1: &Circle, c2: &Circle): bool {
        let d = point::dist_squared(&c1.center, &c2.center);
        let r1 = c1.radius;
        let r2 = c2.radius;
        d*d <= r1*r1 + 2*r1*r2 + r2*r2
    }
}
}
```

Constants

Constants are a way of giving a name to shared, static values inside of a `module` or `script`.

The constant's must be known at compilation. The constant's value is stored in the compiled module or script. And each time the constant is used, a new copy of that value is made.

Declaration

Constant declarations begin with the `const` keyword, followed by a name, a type, and a value. They can exist in either a script or module

```
const <name>: <type> = <expression>;
```

For example

```
script {

    const MY_ERROR_CODE: u64 = 0;

    fun main(input: u64) {
        assert!(input > 0, MY_ERROR_CODE);
    }

}

address 0x42 {
module example {

    const MY_ADDRESS: address = @0x42;

    public fun permissioned(s: &signer) {
        assert!(std::signer::address_of(s) == MY_ADDRESS, 0);
    }

}
}
```

Naming

Constants must start with a capital letter `A` to `Z`. After the first letter, constant names can contain underscores `_`, letters `a` to `z`, letters `A` to `Z`, or digits `0` to `9`.

```
const FLAG: bool = false;
const MY_ERROR_CODE: u64 = 0;
const ADDRESS_42: address = @0x42;
```

Even though you can use letters `a` to `z` in a constant. The general style guidelines are to use just uppercase letters `A` to `Z`, with underscores `_` between each word.

This naming restriction of starting with `A` to `Z` is in place to give room for future language features. It may or may not be removed later.

Visibility

`public` constants are not currently supported. `const` values can be used only in the declaring module.

Valid Expressions

Currently, constants are limited to the primitive types `bool`, `u8`, `u16`, `u32`, `u64`, `u128`, `u256`, `address`, and `vector<u8>`. Future support for other `vector` values (besides the "string"-style literals) will come later.

Values

Commonly, `const`s are assigned a simple value, or literal, of their type. For example

```
const MY_BOOL: bool = false;
const MY_ADDRESS: address = @0x70DD;
const BYTES: vector<u8> = b"hello world";
const HEX_BYTES: vector<u8> = x"DEADBEEF";
```

Complex Expressions

In addition to literals, constants can include more complex expressions, as long as the compiler is able to reduce the expression to a value at compile time.

Currently, equality operations, all boolean operations, all bitwise operations, and all arithmetic operations can be used.

```
const RULE: bool = true && false;
const CAP: u64 = 10 * 100 + 1;
const SHIFTY: u8 = {
  (1 << 1) * (1 << 2) * (1 << 3) * (1 << 4)
};
const HALF_MAX: u128 = 340282366920938463463374607431768211455 / 2;
const REM: u256 =
57896044618658097711785492504343953926634992332820282019728792003956564819968 % 654321;
const EQUAL: bool = 1 == 1;
```

If the operation would result in a runtime exception, the compiler will give an error that it is unable to generate the constant's value

```
const DIV_BY_ZERO: u64 = 1 / 0; // error!
const SHIFT_BY_A_LOT: u64 = 1 << 100; // error!
const NEGATIVE_U64: u64 = 0 - 1; // error!
```

Note that constants cannot currently refer to other constants. This feature, along with support for other expressions, will be added in the future.

Generics

Generics can be used to define functions and structs over different input data types. This language feature is sometimes referred to as *parametric polymorphism*. In Move, we will often use the term generics interchangeably with type parameters and type arguments.

Generics are commonly used in library code, such as in vector, to declare code that works over any possible instantiation (that satisfies the specified constraints). In other frameworks, generic code can sometimes be used to interact with global storage many different ways that all still share the same implementation.

Declaring Type Parameters

Both functions and structs can take a list of type parameters in their signatures, enclosed by a pair of angle brackets `<...>`.

Generic Functions

Type parameters for functions are placed after the function name and before the (value) parameter list. The following code defines a generic identity function that takes a value of any type and returns that value unchanged.

```
fun id<T>(x: T): T {
    // this type annotation is unnecessary but valid
    (x: T)
}
```

Once defined, the type parameter `T` can be used in parameter types, return types, and inside the function body.

Generic Structs

Type parameters for structs are placed after the struct name, and can be used to name the types of the fields.

```
struct Foo<T> has copy, drop { x: T }

struct Bar<T1, T2> has copy, drop {
    x: T1,
    y: vector<T2>,
}
```

Note that type parameters do not have to be used

Type Arguments

Calling Generic Functions

When calling a generic function, one can specify the type arguments for the function's type parameters in a list enclosed by a pair of angle brackets.

```
fun foo() {
    let x = id<bool>(true);
}
```

If you do not specify the type arguments, Move's type inference will supply them for you.

Using Generic Structs

Similarly, one can attach a list of type arguments for the struct's type parameters when constructing or destructing values of generic types.

```
fun foo() {
    let foo = Foo<bool> { x: true };
    let Foo<bool> { x } = foo;
}
```

If you do not specify the type arguments, Move's type inference will supply them for you.

Type Argument Mismatch

If you specify the type arguments and they conflict with the actual values supplied, an error will be given:

```
fun foo() {
    let x = id<u64>(true); // error! true is not a u64
}
```

and similarly:

```
fun foo() {
    let foo = Foo<bool> { x: 0 }; // error! 0 is not a bool
    let Foo<address> { x } = foo; // error! bool is incompatible with address
}
```

Type Inference

In most cases, the Move compiler will be able to infer the type arguments so you don't have to write them down explicitly. Here's what the examples above would look like if we omit the type arguments:

```
fun foo() {
    let x = id(true);
    //        ^ <bool> is inferred

    let foo = Foo { x: true };
    //           ^ <bool> is inferred

    let Foo { x } = foo;
    //     ^ <bool> is inferred
}
```

Note: when the compiler is unable to infer the types, you'll need annotate them manually. A common scenario is to call a function with type parameters appearing only at return positions.

```
address 0x2 {
module m {
    using std::vector;

    fun foo() {
        // let v = vector::new();
        //                   ^ The compiler cannot figure out the element type.

        let v = vector::new<u64>();
        //                 ^~~~~ Must annotate manually.
    }
}
}
```

However, the compiler will be able to infer the type if that return value is used later in that function:

```
address 0x2 {
module m {
    using std::vector;

    fun foo() {
        let v = vector::new();
        //                  ^ <u64> is inferred
        vector::push_back(&mut v, 42);
    }
}
}
```

Unused Type Parameters

For a struct definition, an unused type parameter is one that does not appear in any field defined in the struct, but is checked statically at compile time. Move allows unused type parameters so the following struct definition is valid:

```
struct Foo<T> {
    foo: u64
}
```

This can be convenient when modeling certain concepts. Here is an example:

```
address 0x2 {
module m {
    // Currency Specifiers
    struct Currency1 {}
    struct Currency2 {}

    // A generic coin type that can be instantiated using a currency
    // specifier type.
    //   e.g. Coin<Currency1>, Coin<Currency2> etc.
    struct Coin<Currency> has store {
        value: u64
    }

    // Write code generically about all currencies
    public fun mint_generic<Currency>(value: u64): Coin<Currency> {
        Coin { value }
    }

    // Write code concretely about one currency
    public fun mint_concrete(value: u64): Coin<Currency1> {
        Coin { value }
    }
}
}
```

In this example, `struct Coin<Currency>` is generic on the `Currency` type parameter, which specifies the currency of the coin and allows code to be written either generically on any currency or concretely on a specific currency. This genericity applies even when the `Currency` type parameter does not appear in any of the fields defined in `Coin`.

Phantom Type Parameters

In the example above, although `struct Coin` asks for the `store` ability, neither `Coin<Currency1>` nor `Coin<Currency2>` will have the `store` ability. This is because of the rules for Conditional

Abilities and Generic Types and the fact that `Currency1` and `Currency2` don't have the `store` ability, despite the fact that they are not even used in the body of `struct Coin`. This might cause some unpleasant consequences. For example, we are unable to put `Coin<Currency1>` into a wallet in the global storage.

One possible solution would be to add spurious ability annotations to `Currency1` and `Currency2` (i.e., `struct Currency1 has store {}`). But, this might lead to bugs or security vulnerabilities because it weakens the types with unnecessary ability declarations. For example, we would never expect a resource in the global storage to have a field in type `Currency1`, but this would be possible with the spurious `store` ability. Moreover, the spurious annotations would be infectious, requiring many functions generic on the unused type parameter to also include the necessary constraints.

Phantom type parameters solve this problem. Unused type parameters can be marked as *phantom* type parameters, which do not participate in the ability derivation for structs. In this way, arguments to phantom type parameters are not considered when deriving the abilities for generic types, thus avoiding the need for spurious ability annotations. For this relaxed rule to be sound, Move's type system guarantees that a parameter declared as `phantom` is either not used at all in the struct definition, or it is only used as an argument to type parameters also declared as `phantom`.

Declaration

In a struct definition a type parameter can be declared as phantom by adding the `phantom` keyword before its declaration. If a type parameter is declared as phantom we say it is a phantom type parameter. When defining a struct, Move's type checker ensures that every phantom type parameter is either not used inside the struct definition or it is only used as an argument to a phantom type parameter.

More formally, if a type is used as an argument to a phantom type parameter we say the type appears in *phantom position*. With this definition in place, the rule for the correct use of phantom parameters can be specified as follows: **A phantom type parameter can only appear in phantom position**.

The following two examples show valid uses of phantom parameters. In the first one, the parameter `T1` is not used at all inside the struct definition. In the second one, the parameter `T1` is only used as an argument to a phantom type parameter.

```
struct S1<phantom T1, T2> { f: u64 }
                  ^^
                  Ok: T1 does not appear inside the struct definition


struct S2<phantom T1, T2> { f: S1<T1, T2> }
                                  ^^
                                  Ok: T1 appears in phantom position
```

The following code shows examples of violations of the rule:

```
struct S1<phantom T> { f: T }
                          ^
                          Error: Not a phantom position

struct S2<T> { f: T }

struct S3<phantom T> { f: S2<T> }
                             ^
                             Error: Not a phantom position
```

Instantiation

When instantiating a struct, the arguments to phantom parameters are excluded when deriving the struct abilities. For example, consider the following code:

```
struct S<T1, phantom T2> has copy { f: T1 }
struct NoCopy {}
struct HasCopy has copy {}
```

Consider now the type `S<HasCopy, NoCopy>`. Since `S` is defined with `copy` and all non-phantom arguments have `copy` then `S<HasCopy, NoCopy>` also has `copy`.

Phantom Type Parameters with Ability Constraints

Ability constraints and phantom type parameters are orthogonal features in the sense that phantom parameters can be declared with ability constraints. When instantiating a phantom type parameter with an ability constraint, the type argument has to satisfy that constraint, even though the parameter is phantom. For example, the following definition is perfectly valid:

```
struct S<phantom T: copy> {}
```

The usual restrictions apply and `T` can only be instantiated with arguments having `copy`.

Constraints

In the examples above, we have demonstrated how one can use type parameters to define "unknown" types that can be plugged in by callers at a later time. This however means the type system has little information about the type and has to perform checks in a very conservative way. In some sense, the type system must assume the worst case scenario for an unconstrained generic. Simply put, by default generic type parameters have no abilities.

This is where constraints come into play: they offer a way to specify what properties these unknown types have so the type system can allow operations that would otherwise be unsafe.

Declaring Constraints

Constraints can be imposed on type parameters using the following syntax.

```
// T is the name of the type parameter
T: <ability> (+ <ability>)*
```

The `<ability>` can be any of the four abilities, and a type parameter can be constrained with multiple abilities at once. So all of the following would be valid type parameter declarations:

```
T: copy
T: copy + drop
T: copy + drop + store + key
```

Verifying Constraints

Constraints are checked at call sites so the following code won't compile.

```
struct Foo<T: key> { x: T }

struct Bar { x: Foo<u8> }
//                  ^ error! u8 does not have 'key'

struct Baz<T> { x: Foo<T> }
//                     ^ error! T does not have 'key'
```

```
struct R {}

fun unsafe_consume<T>(x: T) {
    // error! x does not have 'drop'
}

fun consume<T: drop>(x: T) {
    // valid!
    // x will be dropped automatically
}

fun foo() {
    let r = R {};
    consume<R>(r);
    //      ^ error! R does not have 'drop'
}
```

```
struct R {}

fun unsafe_double<T>(x: T) {
    (copy x, x)
    // error! x does not have 'copy'
}

fun double<T: copy>(x: T) {
    (copy x, x) // valid!
}

fun foo(): (R, R) {
    let r = R {};
    double<R>(r)
    //     ^ error! R does not have 'copy'
}
```

For more information, see the abilities section on conditional abilities and generic types.

Limitations on Recursions

Recursive Structs

Generic structs can not contain fields of the same type, either directly or indirectly, even with different type arguments. All of the following struct definitions are invalid:

```
struct Foo<T> {
    x: Foo<u64> // error! 'Foo' containing 'Foo'
}

struct Bar<T> {
    x: Bar<T> // error! 'Bar' containing 'Bar'
}

// error! 'A' and 'B' forming a cycle, which is not allowed either.
struct A<T> {
    x: B<T, u64>
}

struct B<T1, T2> {
    x: A<T1>
    y: A<T2>
}
```

Advanced Topic: Type-level Recursions

Move allows generic functions to be called recursively. However, when used in combination with generic structs, this could create an infinite number of types in certain cases, and allowing this means adding unnecessary complexity to the compiler, vm and other language components. Therefore, such recursions are forbidden.

Allowed:

```
address 0x2 {
module m {
    struct A<T> {}

    // Finitely many types -- allowed.
    // foo<T> -> foo<T> -> foo<T> -> ... is valid
    fun foo<T>() {
        foo<T>();
    }

    // Finitely many types -- allowed.
    // foo<T> -> foo<A<u64>> -> foo<A<u64>> -> ... is valid
    fun foo<T>() {
        foo<A<u64>>();
    }
}
}
```

Not allowed:

```
address 0x2 {
module m {
    struct A<T> {}

    // Infinitely many types -- NOT allowed.
    // error!
    // foo<T> -> foo<A<T>> -> foo<A<A<T>>> -> ...
    fun foo<T>() {
        foo<Foo<T>>();
    }
}
}
```

```
address 0x2 {
module n {
    struct A<T> {}

    // Infinitely many types -- NOT allowed.
    // error!
    // foo<T1, T2> -> bar<T2, T1> -> foo<T2, A<T1>>
    //   -> bar<A<T1>, T2> -> foo<A<T1>, A<T2>>
    //   -> bar<A<T2>, A<T1>> -> foo<A<T2>, A<A<T1>>>
    //   -> ...
    fun foo<T1, T2>() {
        bar<T2, T1>();
    }

    fun bar<T1, T2> {
        foo<T1, A<T2>>();
    }
}
}
```

Note, the check for type level recursions is based on a conservative analysis on the call sites and does NOT take control flow or runtime values into account.

```
address 0x2 {
module m {
    struct A<T> {}

    fun foo<T>(n: u64) {
        if (n > 0) {
            foo<A<T>>(n - 1);
        };
    }
}
}
```

The function in the example above will technically terminate for any given input and therefore only creating finitely many types, but it is still considered invalid by Move's type system.

Abilities

Abilities are a typing feature in Move that control what actions are permissible for values of a given type. This system grants fine grained control over the "linear" typing behavior of values, as well as if and how values are used in global storage. This is implemented by gating access to certain bytecode instructions so that for a value to be used with the bytecode instruction, it must have the ability required (if one is required at all—not every instruction is gated by an ability).

The Four Abilities

The four abilities are:

- `copy`
  - Allows values of types with this ability to be copied.
- `drop`
  - Allows values of types with this ability to be popped/dropped.
- `store`
  - Allows values of types with this ability to exist inside a struct in global storage.
- `key`
  - Allows the type to serve as a key for global storage operations.

`copy`

The `copy` ability allows values of types with that ability to be copied. It gates the ability to copy values out of local variables with the `copy` operator and to copy values via references with dereference `*e`.

If a value has `copy`, all values contained inside of that value have `copy`.

`drop`

The `drop` ability allows values of types with that ability to be dropped. By dropped, we mean that value is not transferred and is effectively destroyed as the Move program executes. As such, this ability gates the ability to ignore values in a multitude of locations, including:

- not using the value in a local variable or parameter
- not using the value in a sequence via `;`
- overwriting values in variables in assignments
- overwriting values via references when writing `*e1 = e2`.

If a value has `drop`, all values contained inside of that value have `drop`.

`store`

The `store` ability allows values of types with this ability to exist inside of a struct (resource) in global storage, *but* not necessarily as a top-level resource in global storage. This is the only ability that does not directly gate an operation. Instead it gates the existence in global storage when used in tandem with `key`.

If a value has `store`, all values contained inside of that value have `store`

`key`

The `key` ability allows the type to serve as a key for global storage operations. It gates all global storage operations, so in order for a type to be used with `move_to`, `borrow_global`, `move_from`,

etc., the type must have the `key` ability. Note that the operations still must be used in the module where the `key` type is defined (in a sense, the operations are private to the defining module).

If a value has `key`, all values contained inside of that value have `store`. This is the only ability with this sort of asymmetry.

Builtin Types

Most primitive, builtin types have `copy`, `drop`, and `store` with the exception of `signer`, which just has `drop`

- `bool`, `u8`, `u16`, `u32`, `u64`, `u128`, `u256`, and `address` all have `copy`, `drop`, and `store`.
- `signer` has `drop`
  - Cannot be copied and cannot be put into global storage
- `vector<T>` may have `copy`, `drop`, and `store` depending on the abilities of `T`.
  - See Conditional Abilities and Generic Types for more details.
- Immutable references `&` and mutable references `&mut` both have `copy` and `drop`.
  - This refers to copying and dropping the reference itself, not what they refer to.
  - References cannot appear in global storage, hence they do not have `store`.

None of the primitive types have `key`, meaning none of them can be used directly with the global storage operations.

Annotating Structs

To declare that a `struct` has an ability, it is declared with `has <ability>` after the struct name but before the fields. For example:

```
struct Ignorable has drop { f: u64 }
struct Pair has copy, drop, store { x: u64, y: u64 }
```

In this case: `Ignorable` has the `drop` ability. `Pair` has `copy`, `drop`, and `store`.

All of these abilities have strong guarantees over these gated operations. The operation can be performed on the value only if it has that ability; even if the value is deeply nested inside of some other collection!

As such: when declaring a struct's abilities, certain requirements are placed on the fields. All fields must satisfy these constraints. These rules are necessary so that structs satisfy the reachability rules for the abilities given above. If a struct is declared with the ability...

- `copy`, all fields must have `copy`.
- `drop`, all fields must have `drop`.
- `store`, all fields must have `store`.
- `key`, all fields must have `store`.
    - `key` is the only ability currently that doesn't require itself.

For example:

```
// A struct without any abilities
struct NoAbilities {}

struct WantsCopy has copy {
    f: NoAbilities, // ERROR 'NoAbilities' does not have 'copy'
}
```

and similarly:

```
// A struct without any abilities
struct NoAbilities {}

struct MyResource has key {
    f: NoAbilities, // Error 'NoAbilities' does not have 'store'
}
```

Conditional Abilities and Generic Types

When abilities are annotated on a generic type, not all instances of that type are guaranteed to have that ability. Consider this struct declaration:

```
struct Cup<T> has copy, drop, store, key { item: T }
```

It might be very helpful if `Cup` could hold any type, regardless of its abilities. The type system can *see* the type parameter, so it should be able to remove abilities from `Cup` if it *sees* a type parameter that would violate the guarantees for that ability.

This behavior might sound a bit confusing at first, but it might be more understandable if we think about collection types. We could consider the builtin type `vector` to have the following type declaration:

```
vector<T> has copy, drop, store;
```

We want `vector`s to work with any type. We don't want separate `vector` types for different abilities. So what are the rules we would want? Precisely the same that we would want with the field rules above. So, it would be safe to copy a `vector` value only if the inner elements can be copied. It would be safe to ignore a `vector` value only if the inner elements can be ignored/dropped. And, it would be safe to put a `vector` in global storage only if the inner elements can be in global storage.

To have this extra expressiveness, a type might not have all the abilities it was declared with depending on the instantiation of that type; instead, the abilities a type will have depends on both its declaration **and** its type arguments. For any type, type parameters are pessimistically assumed to be used inside of the struct, so the abilities are only granted if the type parameters meet the requirements described above for fields. Taking `Cup` from above as an example:

- `Cup` has the ability `copy` only if `T` has `copy`.
- It has `drop` only if `T` has `drop`.
- It has `store` only if `T` has `store`.
- It has `key` only if `T` has `store`.

Here are examples for this conditional system for each ability:

Example: conditional `copy`

```
struct NoAbilities {}
struct S has copy, drop { f: bool }
struct Cup<T> has copy, drop, store { item: T }

fun example(c_x: Cup<u64>, c_s: Cup<S>) {
    // Valid, 'Cup<u64>' has 'copy' because 'u64' has 'copy'
    let c_x2 = copy c_x;
    // Valid, 'Cup<S>' has 'copy' because 'S' has 'copy'
    let c_s2 = copy c_s;
}

fun invalid(c_account: Cup<signer>, c_n: Cup<NoAbilities>) {
    // Invalid, 'Cup<signer>' does not have 'copy'.
    // Even though 'Cup' was declared with copy, the instance does not have 'copy'
    // because 'signer' does not have 'copy'
    let c_account2 = copy c_account;
    // Invalid, 'Cup<NoAbilities>' does not have 'copy'
    // because 'NoAbilities' does not have 'copy'
    let c_n2 = copy c_n;
}
```

Example: conditional `drop`

```
struct NoAbilities {}
struct S has copy, drop { f: bool }
struct Cup<T> has copy, drop, store { item: T }

fun unused() {
    Cup<bool> { item: true }; // Valid, 'Cup<bool>' has 'drop'
    Cup<S> { item: S { f: false }}; // Valid, 'Cup<S>' has 'drop'
}

fun left_in_local(c_account: Cup<signer>): u64 {
    let c_b = Cup<bool> { item: true };
    let c_s = Cup<S> { item: S { f: false }};
    // Valid return: 'c_account', 'c_b', and 'c_s' have values
    // but 'Cup<signer>', 'Cup<bool>', and 'Cup<S>' have 'drop'
    0
}

fun invalid_unused() {
    // Invalid, Cannot ignore 'Cup<NoAbilities>' because it does not have 'drop'.
    // Even though 'Cup' was declared with 'drop', the instance does not have 'drop'
    // because 'NoAbilities' does not have 'drop'
    Cup<NoAbilities> { item: NoAbilities {}};
}

fun invalid_left_in_local(): u64 {
    let n = Cup<NoAbilities> { item: NoAbilities {}};
    // Invalid return: 'n' has a value
    // and 'Cup<NoAbilities>' does not have 'drop'
    0
}
```

Example: conditional `store`

```
struct Cup<T> has copy, drop, store { item: T }

// 'MyInnerResource' is declared with 'store' so all fields need 'store'
struct MyInnerResource has store {
    yes: Cup<u64>, // Valid, 'Cup<u64>' has 'store'
    // no: Cup<signer>, Invalid, 'Cup<signer>' does not have 'store'
}

// 'MyResource' is declared with 'key' so all fields need 'store'
struct MyResource has key {
    yes: Cup<u64>, // Valid, 'Cup<u64>' has 'store'
    inner: Cup<MyInnerResource>, // Valid, 'Cup<MyInnerResource>' has 'store'
    // no: Cup<signer>, Invalid, 'Cup<signer>' does not have 'store'
}
```

Example: conditional `key`

```
struct NoAbilities {}
struct MyResource<T> has key { f: T }

fun valid(account: &signer) acquires MyResource {
    let addr = signer::address_of(account);
     // Valid, 'MyResource<u64>' has 'key'
    let has_resource = exists<MyResource<u64>>(addr);
    if (!has_resource) {
         // Valid, 'MyResource<u64>' has 'key'
        move_to(account, MyResource<u64> { f: 0 })
    };
    // Valid, 'MyResource<u64>' has 'key'
    let r = borrow_global_mut<MyResource<u64>>(addr)
    r.f = r.f + 1;
}

fun invalid(account: &signer) {
   // Invalid, 'MyResource<NoAbilities>' does not have 'key'
   let has_it = exists<MyResource<NoAbilities>>(addr);
   // Invalid, 'MyResource<NoAbilities>' does not have 'key'
   let NoAbilities {} = move_from<NoAbilities>(addr);
   // Invalid, 'MyResource<NoAbilities>' does not have 'key'
   move_to(account, NoAbilities {});
   // Invalid, 'MyResource<NoAbilities>' does not have 'key'
   borrow_global<NoAbilities>(addr);
}
```

Uses and Aliases

The `use` syntax can be used to create aliases to members in other modules. `use` can be used to create aliases that last either for the entire module, or for a given expression block scope.

Syntax

There are several different syntax cases for `use`. Starting with the most simple, we have the following for creating aliases to other modules

```
use <address>::<module name>;
use <address>::<module name> as <module alias name>;
```

For example

```
use std::vector;
use std::vector as V;
```

`use std::vector;` introduces an alias `vector` for `std::vector`. This means that anywhere you would want to use the module name `std::vector` (assuming this `use` is in scope), you could use `vector` instead. `use std::vector;` is equivalent to `use std::vector as vector;`

Similarly `use std::vector as V;` would let you use `V` instead of `std::vector`

```
use std::vector;
use std::vector as V;

fun new_vecs(): (vector<u8>, vector<u8>, vector<u8>) {
    let v1 = std::vector::empty();
    let v2 = vector::empty();
    let v3 = V::empty();
    (v1, v2, v3)
}
```

If you want to import a specific module member (such as a function, struct, or constant). You can use the following syntax.

```
use <address>::<module name>::<module member>;
use <address>::<module name>::<module member> as <member alias>;
```

For example

```
use std::vector::empty;
use std::vector::empty as empty_vec;
```

This would let you use the function `std::vector::empty` without full qualification. Instead you could use `empty` and `empty_vec` respectively. Again, `use std::vector::empty;` is equivalent to `use std::vector::empty as empty;`

```
use std::vector::empty;
use std::vector::empty as empty_vec;

fun new_vecs(): (vector<u8>, vector<u8>, vector<u8>) {
    let v1 = std::vector::empty();
    let v2 = empty();
    let v3 = empty_vec();
    (v1, v2, v3)
}
```

If you want to add aliases for multiple module members at once, you can do so with the following syntax

```
use <address>::<module name>::{<module member>, <module member> as <member alias> ...
};
```

For example

```
use std::vector::{push_back, length as len, pop_back};

fun swap_last_two<T>(v: &mut vector<T>) {
    assert!(len(v) >= 2, 42);
    let last = pop_back(v);
    let second_to_last = pop_back(v);
    push_back(v, last);
    push_back(v, second_to_last)
}
```

If you need to add an alias to the Module itself in addition to module members, you can do that in a single `use` using `Self`. `Self` is a member of sorts that refers to the module.

```
use std::vector::{Self, empty};
```

For clarity, all of the following are equivalent:

```
use std::vector;
use std::vector as vector;
use std::vector::Self;
use std::vector::Self as vector;
use std::vector::{Self};
use std::vector::{Self as vector};
```

If needed, you can have as many aliases for any item as you like

```
use std::vector::{
    Self,
    Self as V,
    length,
    length as len,
};

fun pop_twice<T>(v: &mut vector<T>): (T, T) {
    // all options available given the `use` above
    assert!(vector::length(v) > 1, 42);
    assert!(V::length(v) > 1, 42);
    assert!(length(v) > 1, 42);
    assert!(len(v) > 1, 42);

    (vector::pop_back(v), vector::pop_back(v))
}
```

Inside a `module`

Inside of a `module` all `use` declarations are usable regardless of the order of declaration.

```
address 0x42 {
module example {
    use std::vector;

    fun example(): vector<u8> {
        let v = empty();
        vector::push_back(&mut v, 0);
        vector::push_back(&mut v, 10);
        v
    }

    use std::vector::empty;
}
}
```

The aliases declared by `use` in the module usable within that module.

Additionally, the aliases introduced cannot conflict with other module members. See Uniqueness for more details

Inside an expression

You can add `use` declarations to the beginning of any expression block

```
address 0x42 {
module example {

    fun example(): vector<u8> {
        use std::vector::{empty, push_back};

        let v = empty();
        push_back(&mut v, 0);
        push_back(&mut v, 10);
        v
    }
}
}
```

As with `let`, the aliases introduced by `use` in an expression block are removed at the end of that block.

```
address 0x42 {
module example {

    fun example(): vector<u8> {
        let result = {
            use std::vector::{empty, push_back};
            let v = empty();
            push_back(&mut v, 0);
            push_back(&mut v, 10);
            v
        };
        result
    }

}
}
```

Attempting to use the alias after the block ends will result in an error

```
fun example(): vector<u8> {
    let result = {
        use std::vector::{empty, push_back};
        let v = empty();
        push_back(&mut v, 0);
        push_back(&mut v, 10);
        v
    };
    let v2 = empty(); // ERROR!
//           ^^^^^ unbound function 'empty'
    result
}
```

Any `use` must be the first item in the block. If the `use` comes after any expression or `let`, it will result in a parsing error

```
{
    let x = 0;
    use std::vector; // ERROR!
    let v = vector::empty();
}
```

Naming rules

Aliases must follow the same rules as other module members. This means that aliases to structs or constants must start with `A` to `Z`

```
address 0x42 {
module data {
    struct S {}
    const FLAG: bool = false;
    fun foo() {}
}
module example {
    use 0x42::data::{
        S as s, // ERROR!
        FLAG as fLAG, // ERROR!
        foo as FOO,  // valid
        foo as bar, // valid
    };
}
}
```

Uniqueness

Inside a given scope, all aliases introduced by `use` declarations must be unique.

For a module, this means aliases introduced by `use` cannot overlap

```
address 0x42 {
module example {

    use std::vector::{empty as foo, length as foo}; // ERROR!
    //                                  ^^^ duplicate 'foo'

    use std::vector::empty as bar;

    use std::vector::length as bar; // ERROR!
    //                         ^^^ duplicate 'bar'

}
}
```

And, they cannot overlap with any of the module's other members

```
address 0x42 {
module data {
    struct S {}
}
module example {
    use 0x42::data::S;

    struct S { value: u64 } // ERROR!
    //     ^ conflicts with alias 'S' above
}
}
```

Inside of an expression block, they cannot overlap with each other, but they can shadow other aliases or names from an outer scope

Shadowing

`use` aliases inside of an expression block can shadow names (module members or aliases) from the outer scope. As with shadowing of locals, the shadowing ends at the end of the expression block;

```
address 0x42 {
module example {

    struct WrappedVector { vec: vector<u64> }

    fun empty(): WrappedVector {
        WrappedVector { vec: std::vector::empty() }
    }

    fun example1(): (WrappedVector, WrappedVector) {
        let vec = {
            use std::vector::{empty, push_back};
            // 'empty' now refers to std::vector::empty

            let v = empty();
            push_back(&mut v, 0);
            push_back(&mut v, 1);
            push_back(&mut v, 10);
            v
        };
        // 'empty' now refers to Self::empty

        (empty(), WrappedVector { vec })
    }

    fun example2(): (WrappedVector, WrappedVector) {
        use std::vector::{empty, push_back};
        let w: WrappedVector = {
            use 0x42::example::empty;
            empty()
        };
        push_back(&mut w.vec, 0);
        push_back(&mut w.vec, 1);
        push_back(&mut w.vec, 10);

        let vec = empty();
        push_back(&mut vec, 0);
        push_back(&mut vec, 1);
        push_back(&mut vec, 10);
```

```
        (w, WrappedVector { vec })
    }
}
}
```

Unused Use or Alias

An unused `use` will result in an error

```
address 0x42 {
module example {
    use std::vector::{empty, push_back}; // ERROR!
    //                    ^^^^^^^^^ unused alias 'push_back'

    fun example(): vector<u8> {
        empty()
    }
}
}
```

Friends

The `friend` syntax is used to declare modules that are trusted by the current module. A trusted module is allowed to call any function defined in the current module that have the `public(friend)` visibility. For details on function visibilities, please refer to the *Visibility* section in Functions.

Friend declaration

A module can declare other modules as friends via friend declaration statements, in the format of

- `friend <address::name>` — friend declaration using fully qualified module name like the example below, or

```
address 0x42 {
module a {
    friend 0x42::b;
}
}
```

- `friend <module-name-alias>` — friend declaration using a module name alias, where the module alias is introduced via the `use` statement.

```
address 0x42 {
module a {
    use 0x42::b;
    friend b;
}
}
```

A module may have multiple friend declarations, and the union of all the friend modules forms the friend list. In the example below, both `0x42::B` and `0x42::C` are considered as friends of `0x42::A`.

```
address 0x42 {
module a {
    friend 0x42::b;
    friend 0x42::c;
}
}
```

Unlike `use` statements, `friend` can only be declared in the module scope and not in the expression block scope. `friend` declarations may be located anywhere a top-level construct (e.g., `use`, `function`, `struct`, etc.) is allowed. However, for readability, it is advised to place friend declarations near the beginning of the module definition.

Note that the concept of friendship does not apply to Move scripts:

- A Move script cannot declare `friend` modules as doing so is considered meaningless: there is no mechanism to call the function defined in a script.
- A Move module cannot declare `friend` scripts as well because scripts are ephemeral code snippets that are never published to global storage.

Friend declaration rules

Friend declarations are subject to the following rules:

- A module cannot declare itself as a friend.

```
address 0x42 {
module m { friend Self; // ERROR! }
//                ^^^^ Cannot declare the module itself as a friend
}

address 0x43 {
module m { friend 0x43::M; // ERROR! }
//                ^^^^^^^ Cannot declare the module itself as a friend
}
```

- Friend modules must be known by the compiler

```
address 0x42 {
module m { friend 0x42::nonexistent; // ERROR! }
//                ^^^^^^^^^^^^^^^^^ Unbound module '0x42::nonexistent'
}
```

- Friend modules must be within the same account address. (Note: this is not a technical requirement but rather a policy decision which *may* be relaxed later.)

```
address 0x42 {
module m {}
}

address 0x43 {
module n { friend 0x42::m; // ERROR! }
//                ^^^^^^^ Cannot declare modules out of the current address as a
friend
}
```

- Friends relationships cannot create cyclic module dependencies.

  Cycles are not allowed in the friend relationships, e.g., the relation `0x2::a` friends `0x2::b` friends `0x2::c` friends `0x2::a` is not allowed. More generally, declaring a friend module adds a dependency upon the current module to the friend module (because the purpose is for the friend to call functions in the current module). If that friend module is already used, either directly or transitively, a cycle of dependencies would be created.

```
address 0x2 {
module a {
    use 0x2::c;
    friend 0x2::b;

    public fun a() {
        c::c()
    }
}

module b {
    friend 0x2::c; // ERROR!
//         ^^^^^^ This friend relationship creates a dependency cycle: '0x2::b' is
a friend of '0x2::a' uses '0x2::c' is a friend of '0x2::b'
}

module c {
    public fun c() {}
}
}
```

- The friend list for a module cannot contain duplicates.

```
address 0x42 {
module a {}

module m {
    use 0x42::a as aliased_a;
    friend 0x42::A;
    friend aliased_a; // ERROR!
//         ^^^^^^^^^ Duplicate friend declaration '0x42::a'. Friend declarations
in a module must be unique
}
}
```

Packages

Packages allow Move programmers to more easily re-use code and share it across projects. The Move package system allows programmers to easily:

- Define a package containing Move code;
- Parameterize a package by named addresses;
- Import and use packages in other Move code and instantiate named addresses;
- Build packages and generate associated compilation artifacts from packages; and
- Work with a common interface around compiled Move artifacts.

Package Layout and Manifest Syntax

A Move package source directory contains a `Move.toml` package manifest file along with a set of subdirectories:

```
a_move_package
├── Move.toml      (required)
├── sources        (required)
├── examples       (optional, test & dev mode)
├── scripts        (optional)
├── doc_templates  (optional)
└── tests          (optional, test mode)
```

The directories marked `required` *must* be present in order for the directory to be considered a Move package and to be compiled. Optional directories can be present, and if so will be included in the compilation process. Depending on the mode that the package is built with (`test` or `dev`), the `tests` and `examples` directories will be included as well.

The `sources` directory can contain both Move modules and Move scripts (both transaction scripts and modules containing script functions). The `examples` directory can hold additional code to be used only for development and/or tutorial purposes that will not be included when compiled outside `test` or `dev` mode.

A `scripts` directory is supported so transaction scripts can be separated from modules if that is desired by the package author. The `scripts` directory will always be included for compilation if it is present. Documentation will be built using any documentation templates present in the `doc_templates` directory.

Move.toml

The Move package manifest is defined within the `Move.toml` file and has the following syntax. Optional fields are marked with `*`, `+` denotes one or more elements:

```
[package]
name = <string>                  # e.g., "MoveStdlib"
version = "<uint>.<uint>.<uint>" # e.g., "0.1.1"
license* = <string>              # e.g., "MIT", "GPL", "Apache 2.0"
authors* = [<string>]            # e.g., ["Joe Smith (joesmith@noemail.com)", "Jane
Smith (janesmith@noemail.com)"]

[addresses]  # (Optional section) Declares named addresses in this package and
instantiates named addresses in the package graph
# One or more lines declaring named addresses in the following format
<addr_name> = "_" | "<hex_address>" # e.g., std = "_" or my_addr = "0xC0FFEECAFE"

[dependencies] # (Optional section) Paths to dependencies and instantiations or
renamings of named addresses from each dependency
# One or more lines declaring dependencies in the following format
<string> = { local = <string>, addr_subst* = { (<string> = (<string> | "
<hex_address>"))+ } } # local dependencies
<string> = { git = <URL ending in .git>, subdir=<path to dir containing Move.toml
inside git repo>, rev=<git commit hash>, addr_subst* = { (<string> = (<string> | "
<hex_address>"))+ } } # git dependencies

[dev-addresses] # (Optional section) Same as [addresses] section, but only included in
"dev" and "test" modes
# One or more lines declaring dev named addresses in the following format
<addr_name> = "_" | "<hex_address>" # e.g., std = "_" or my_addr = "0xC0FFEECAFE"

[dev-dependencies] # (Optional section) Same as [dependencies] section, but only
included in "dev" and "test" modes
# One or more lines declaring dev dependencies in the following format
<string> = { local = <string>, addr_subst* = { (<string> = (<string> | <address>))+ } }
```

An example of a minimal package manifest with one local dependency and one git dependency:

```
[package]
name = "AName"
version = "0.0.0"
```

An example of a more standard package manifest that also includes the Move standard library and instantiates the named address `Std` from it with the address value `0x1`:

```
[package]
name = "AName"
version = "0.0.0"
license = "Apache 2.0"

[addresses]
address_to_be_filled_in = "_"
specified_address = "0xB0B"

[dependencies]
# Local dependency
LocalDep = { local = "projects/move-awesomeness", addr_subst = { "std" = "0x1" } }
# Git dependency
MoveStdlib = { git = "https://github.com/diem/diem.git", subdir="language/move-stdlib",
rev = "56ab033cc403b489e891424a629e76f643d4fb6b" }

[dev-addresses] # For use when developing this module
address_to_be_filled_in = "0x101010101"
```

Most of the sections in the package manifest are self explanatory, but named addresses can be a bit difficult to understand so it's worth examining them in a bit more detail.

Named Addresses During Compilation

Recall that Move has named addresses and that named addresses cannot be declared in Move. Because of this, until now named addresses and their values needed to be passed to the compiler on the command line. With the Move package system this is no longer needed, and you can declare named addresses in the package, instantiate other named addresses in scope, and rename named addresses from other packages within the Move package system manifest file. Let's go through each of these individually:

Declaration

Let's say we have a Move module in `example_pkg/sources/A.move` as follows:

```
module named_addr::A {
    public fun x(): address { @named_addr }
}
```

We could in `example_pkg/Move.toml` declare the named address `named_addr` in two different ways. The first:

```
[package]
name = "ExamplePkg"
...
[addresses]
named_addr = "_"
```

Declares `named_addr` as a named address in the package `ExamplePkg` and that *this address can be any valid address value*. Therefore an importing package can pick the value of the named address `named_addr` to be any address it wishes. Intuitively you can think of this as parameterizing the package `ExamplePkg` by the named address `named_addr`, and the package can then be instantiated later on by an importing package.

`named_addr` can also be declared as:

```
[package]
name = "ExamplePkg"
...
[addresses]
named_addr = "0xCAFE"
```

which states that the named address `named_addr` is exactly `0xCAFE` and cannot be changed. This is useful so other importing packages can use this named address without needing to worry about the exact value assigned to it.

With these two different declaration methods, there are two ways that information about named addresses can flow in the package graph:

- The former ("unassigned named addresses") allows named address values to flow from the importation site to the declaration site.
- The latter ("assigned named addresses") allows named address values to flow from the declaration site upwards in the package graph to usage sites.

With these two methods for flowing named address information throughout the package graph the rules around scoping and renaming become important to understand.

Scoping and Renaming of Named Addresses

A named address `N` in a package `P` is in scope if:

1. It declares a named address `N`; or
2. A package in one of `P`'s transitive dependencies declares the named address `N` and there is a dependency path in the package graph between between `P` and the declaring package of `N` with no renaming of `N`.

Additionally, every named address in a package is exported. Because of this and the above scoping rules each package can be viewed as coming with a set of named addresses that will be brought into scope when the package is imported, e.g., if the `ExamplePkg` package was imported, that importation would bring into scope the `named_addr` named address. Because of this, if `P` imports two packages `P1` and `P2` both of which declare a named address `N` an issue arises in `P`: which "`N`" is meant when `N` is referred to in `P`? The one from `P1` or `P2`? To prevent this ambiguity around which package a named address is coming from, we enforce that the sets of scopes introduced by all dependencies in a package are disjoint, and provide a way to *rename named addresses* when the package that brings them into scope is imported.

Renaming a named address when importing can be done as follows in our `P`, `P1`, and `P2` example above:

```
[package]
name = "P"
...
[dependencies]
P1 = { local = "some_path_to_P1", addr_subst = { "P1N" = "N" } }
P2 = { local = "some_path_to_P2"  }
```

With this renaming `N` refers to the `N` from `P2` and `P1N` will refer to `N` coming from `P1`:

```
module N::A {
    public fun x(): address { @P1N }
}
```

It is important to note that *renaming is not local*: once a named address `N` has been renamed to `N2` in a package `P` all packages that import `P` will not see `N` but only `N2` unless `N` is reintroduced from outside of `P`. This is why rule (2) in the scoping rules at the start of this section specifies a "dependency path in the package graph between between `P` and the declaring package of `N` with no renaming of `N`."

Instantiation

Named addresses can be instantiated multiple times across the package graph as long as it is always with the same value. It is an error if the same named address (regardless of renaming) is instantiated with differing values across the package graph.

A Move package can only be compiled if all named addresses resolve to a value. This presents issues if the package wishes to expose an uninstantiated named address. This is what the `[dev-addresses]` section solves. This section can set values for named addresses, but cannot introduce any named addresses. Additionally, only the `[dev-addresses]` in the root package are included in

`dev` mode. For example a root package with the following manifest would not compile outside of `dev` mode since `named_addr` would be uninstantiated:

```
[package]
name = "ExamplePkg"
...
[addresses]
named_addr = "_"

[dev-addresses]
named_addr = "0xC0FFEE"
```

Usage, Artifacts, and Data Structures

The Move package system comes with a command line option as part of the Move CLI `move <flags> <command> <command_flags>`. Unless a particular path is provided, all package commands will run in the current working directory. The full list of commands and flags for the Move CLI can be found by running `move --help`.

Usage

A package can be compiled either through the Move CLI commands, or as a library command in Rust with the function `compile_package`. This will create a `CompiledPackage` that holds the compiled bytecode along with other compilation artifacts (source maps, documentation, ABIs) in memory. This `CompiledPackage` can be converted to an `OnDiskPackage` and vice versa -- the latter being the data of the `CompiledPackage` laid out in the file system in the following format:

```
a_move_package
├── Move.toml
...
└── build
    ├── <dep_pkg_name>
    │   ├── BuildInfo.yaml
    │   ├── bytecode_modules
    │   │   └── *.mv
    │   ├── source_maps
    │   │   └── *.mvsm
    │   ├── bytecode_scripts
    │   │   └── *.mv
    │   ├── abis
    │   │   ├── *.abi
    │   │   └── <module_name>/*.abi
    │   └── sources
    │       └── *.move
    │
    ...
    └── <dep_pkg_name>
        ├── BuildInfo.yaml
        ...
        └── sources
```

See the `move-package` crate for more information on these data structures and how to use the Move package system as a Rust library.

Unit Tests

Unit testing for Move adds three new annotations to the Move source language:

- `#[test]`
- `#[test_only]`, and
- `#[expected_failure]`.

They respectively mark a function as a test, mark a module or module member (`use`, function, or struct) as code to be included for testing only, and mark that a test is expected to fail. These annotations can be placed on a function with any visibility. Whenever a module or module member is annotated as `#[test_only]` or `#[test]`, it will not be included in the compiled bytecode unless it is compiled for testing.

Testing Annotations: Their Meaning and Usage

Both the `#[test]` and `#[expected_failure]` annotations can be used either with or without arguments.

Without arguments, the `#[test]` annotation can only be placed on a function with no parameters. This annotation simply marks this function as a test to be run by the unit testing harness.

```
#[test] // OK
fun this_is_a_test() { ... }

#[test] // Will fail to compile since the test takes an argument
fun this_is_not_correct(arg: signer) { ... }
```

A test can also be annotated as an `#[expected_failure]`. This annotation marks that the test should is expected to raise an error. You can ensure that a test is aborting with a specific abort code by annotating it with `#[expected_failure(abort_code = <code>)]`, if it then fails with a different abort code or with a non-abort error the test will fail. Only functions that have the `#[test]` annotation can also be annotated as an `#[expected_failure]`.

```
#[test]
#[expected_failure]
public fun this_test_will_abort_and_pass() { abort 1 }

#[test]
#[expected_failure]
public fun test_will_error_and_pass() { 1/0; }

#[test]
#[expected_failure(abort_code = 0)]
public fun test_will_error_and_fail() { 1/0; }

#[test, expected_failure] // Can have multiple in one attribute. This test will pass.
public fun this_other_test_will_abort_and_pass() { abort 1 }
```

With arguments, a test annotation takes the form `#[test(<param_name_1> = <address>, ..., <param_name_n> = <address>)]`. If a function is annotated in such a manner, the function's parameters must be a permutation of the parameters `<param_name_1>, ..., <param_name_n>`, i.e., the order of these parameters as they occur in the function and their order in the test annotation do not have to be the same, but they must be able to be matched up with each other by name.

Only parameters with a type of `signer` are supported as test parameters. If a non-`signer` parameter is supplied, the test will result in an error when run.

```
#[test(arg = @0xC0FFEE)] // OK
fun this_is_correct_now(arg: signer) { ... }

#[test(wrong_arg_name = @0xC0FFEE)] // Not correct: arg name doesn't match
fun this_is_incorrect(arg: signer) { ... }

#[test(a = @0xC0FFEE, b = @0xCAFE)] // OK. We support multiple signer arguments, but
you must always provide a value for that argument
fun this_works(a: signer, b: signer) { ... }

// somewhere a named address is declared
#[test_only] // test-only named addresses are supported
address TEST_NAMED_ADDR = @0x1;
...
#[test(arg = @TEST_NAMED_ADDR)] // Named addresses are supported!
fun this_is_correct_now(arg: signer) { ... }
```

An expected failure annotation can also take the form `#[expected_failure(abort_code = <u64>)]`. If a test function is annotated in such a way, the test must abort with an abort code equal to `<u64>`. Any other failure or abort code will result in a test failure.

```
#[test, expected_failure(abort_code = 1)] // This test will fail
fun this_test_should_abort_and_fail() { abort 0 }

#[test]
#[expected_failure(abort_code = 0)] // This test will pass
fun this_test_should_abort_and_pass_too() { abort 0 }
```

A module and any of its members can be declared as test only. In such a case the item will only be included in the compiled Move bytecode when compiled in test mode. Additionally, when compiled outside of test mode, any non-test `use`s of a `#[test_only]` module will raise an error during compilation.

```
#[test_only] // test only attributes can be attached to modules
module abc { ... }

#[test_only] // test only attributes can be attached to named addresses
address ADDR = @0x1;

#[test_only] // .. to uses
use 0x1::some_other_module;

#[test_only] // .. to structs
struct SomeStruct { ... }

#[test_only] // .. and functions. Can only be called from test code, but not a test
fun test_only_function(...) { ... }
```

Running Unit Tests

Unit tests for a Move package can be run with the `move test` command.

When running tests, every test will either `PASS`, `FAIL`, or `TIMEOUT`. If a test case fails, the location of the failure along with the function name that caused the failure will be reported if possible. You can see an example of this below.

A test will be marked as timing out if it exceeds the maximum number of instructions that can be executed for any single test. This bound can be changed using the options below, and its default value is set to 5000 instructions. Additionally, while the result of a test is always deterministic, tests are run in parallel by default, so the ordering of test results in a test run is non-deterministic unless running with only one thread (see `OPTIONS` below).

There are also a number of options that can be passed to the unit testing binary to fine-tune testing and to help debug failing tests. These can be found using the the help flag:

```
$ move -h
```

Example

A simple module using some of the unit testing features is shown in the following example:

First create an empty package and change directory into it:

```
$ move new TestExample; cd TestExample
```

Next add the following to the `Move.toml`:

```
[dependencies]
MoveStdlib = { git = "https://github.com/diem/diem.git", subdir="language/move-stdlib",
rev = "56ab033cc403b489e891424a629e76f643d4fb6b", addr_subst = { "std" = "0x1" } }
```

Next add the following module under the `sources` directory:

```
// filename: sources/my_module.move
module 0x1::my_module {

    struct MyCoin has key { value: u64 }

    public fun make_sure_non_zero_coin(coin: MyCoin): MyCoin {
        assert!(coin.value > 0, 0);
        coin
    }

    public fun has_coin(addr: address): bool {
        exists<MyCoin>(addr)
    }

    #[test]
    fun make_sure_non_zero_coin_passes() {
        let coin = MyCoin { value: 1 };
        let MyCoin { value: _ } = make_sure_non_zero_coin(coin);
    }

    #[test]
    // Or #[expected_failure] if we don't care about the abort code
    #[expected_failure(abort_code = 0)]
    fun make_sure_zero_coin_fails() {
        let coin = MyCoin { value: 0 };
        let MyCoin { value: _ } = make_sure_non_zero_coin(coin);
    }

    #[test_only] // test only helper function
    fun publish_coin(account: &signer) {
        move_to(account, MyCoin { value: 1 })
    }

    #[test(a = @0x1, b = @0x2)]
    fun test_has_coin(a: signer, b: signer) {
        publish_coin(&a);
        publish_coin(&b);
        assert!(has_coin(@0x1), 0);
        assert!(has_coin(@0x2), 1);
```

```
        assert!(!has_coin(@0x3), 1);
    }
}
```

Running Tests

You can then run these tests with the `move test` command:

```
$ move test
BUILDING MoveStdlib
BUILDING TestExample
Running Move unit tests
[ PASS    ] 0x1::my_module::make_sure_non_zero_coin_passes
[ PASS    ] 0x1::my_module::make_sure_zero_coin_fails
[ PASS    ] 0x1::my_module::test_has_coin
Test result: OK. Total tests: 3; passed: 3; failed: 0
```

Using Test Flags

`-f <str>` or `--filter <str>`

This will only run tests whose fully qualified name contains `<str>`. For example if we wanted to only run tests with `"zero_coin"` in their name:

```
$ move test -f zero_coin
CACHED MoveStdlib
BUILDING TestExample
Running Move unit tests
[ PASS    ] 0x1::my_module::make_sure_non_zero_coin_passes
[ PASS    ] 0x1::my_module::make_sure_zero_coin_fails
Test result: OK. Total tests: 2; passed: 2; failed: 0
```

`-i <bound>` or `--gas_used <bound>`

This bounds the amount of gas that can be consumed for any one test to `<bound>`:

```
$ move test -i 0
CACHED MoveStdlib
BUILDING TestExample
Running Move unit tests
[ TIMEOUT ] 0x1::my_module::make_sure_non_zero_coin_passes
[ TIMEOUT ] 0x1::my_module::make_sure_zero_coin_fails
[ TIMEOUT ] 0x1::my_module::test_has_coin

Test failures:

Failures in 0x1::my_module:

┌── make_sure_non_zero_coin_passes ──────
│ Test timed out
└──────────────────


┌── make_sure_zero_coin_fails ──────
│ Test timed out
└──────────────────


┌── test_has_coin ──────
│ Test timed out
└──────────────────

Test result: FAILED. Total tests: 3; passed: 0; failed: 3
```

`-s` or `--statistics`

With these flags you can gather statistics about the tests run and report the runtime and gas used for each test. For example, if we wanted to see the statistics for the tests in the example above:

```
$ move test -s
CACHED MoveStdlib
BUILDING TestExample
Running Move unit tests
[ PASS    ] 0x1::my_module::make_sure_non_zero_coin_passes
[ PASS    ] 0x1::my_module::make_sure_zero_coin_fails
[ PASS    ] 0x1::my_module::test_has_coin

Test Statistics:

┌──────────────────────────────────────────┬────────────┬──────────────────────────┐
│                Test Name                 │    Time    │         Gas Used         │
├──────────────────────────────────────────┼────────────┼──────────────────────────┤
│ 0x1::my_module::make_sure_non_zero_coin_passes │   0.009    │            1             │
├──────────────────────────────────────────┼────────────┼──────────────────────────┤
│ 0x1::my_module::make_sure_zero_coin_fails      │   0.008    │            1             │
├──────────────────────────────────────────┼────────────┼──────────────────────────┤
│ 0x1::my_module::test_has_coin            │   0.008    │            1             │
└──────────────────────────────────────────┴────────────┴──────────────────────────┘

Test result: OK. Total tests: 3; passed: 3; failed: 0
```

`-g` or `--state-on-error`

These flags will print the global state for any test failures. e.g., if we added the following (failing) test to the `my_module` example:

```
module 0x1::my_module {
    ...
    #[test(a = @0x1)]
    fun test_has_coin_bad(a: signer) {
        publish_coin(&a);
        assert!(has_coin(@0x1), 0);
        assert!(has_coin(@0x2), 1);
    }
}
```

we would get the following output when running the tests:

```
$ move test -g
CACHED MoveStdlib
BUILDING TestExample
Running Move unit tests
[ PASS    ] 0x1::my_module::make_sure_non_zero_coin_passes
[ PASS    ] 0x1::my_module::make_sure_zero_coin_fails
[ PASS    ] 0x1::my_module::test_has_coin
[ FAIL    ] 0x1::my_module::test_has_coin_bad

Test failures:

Failures in 0x1::my_module:

┌── test_has_coin_bad ──────
│ error[E11001]: test failure
│     ┌─ /home/tzakian/TestExample/sources/my_module.move:47:10
│     │
│ 44  │         fun test_has_coin_bad(a: signer) {
│     │             ----------------- In this function in 0x1::my_module
│     ·
│ 47  │             assert!(has_coin(@0x2), 1);
│     │             ^^^^^^^^^^^^^^^^^^^^^^^^^^ Test was not expected to abort but it
aborted with 1 here
│
│
│ ──────── Storage state at point of failure ──────
│ 0x1:
│       => key 0x1::my_module::MyCoin {
│           value: 1
│       }
│
└──────────────────

Test result: FAILED. Total tests: 4; passed: 3; failed: 1
```

Global Storage - Structure

The purpose of Move programs is to read from and write to tree-shaped persistent global storage. Programs cannot access the filesystem, network, or any other data outside of this tree.

In pseudocode, the global storage looks something like:

```
struct GlobalStorage {
  resources: Map<(address, ResourceType), ResourceValue>
  modules: Map<(address, ModuleName), ModuleBytecode>
}
```

Structurally, global storage is a forest consisting of trees rooted at an account `address`. Each address can store both resource data values and module code values. As the pseudocode above indicates, each `address` can store at most one resource value of a given type and at most one module with a given name.

Global Storage - Operators

Move programs can create, delete, and update resources in global storage using the following five instructions:

Operation	Description	Aborts?
`move_to<T>(&signer,T)`	Publish `T` under `signer.address`	If `signer.address` already holds a `T`
`move_from<T>(address): T`	Remove `T` from `address` and return it	If `address` does not hold a `T`
`borrow_global_mut<T>(address): &mut T`	Return a mutable reference to the `T` stored under `address`	If `address` does not hold a `T`
`borrow_global<T>(address): &T`	Return an immutable reference to the `T` stored under `address`	If `address` does not hold a `T`
`exists<T>(address): bool`	Return `true` if a `T` is stored under `address`	Never

Each of these instructions is parameterized by a type `T` with the `key` ability. However, each type `T` *must be declared in the current module*. This ensures that a resource can only be manipulated via the API exposed by its defining module. The instructions also take either an `address` or `&signer` representing the account address where the resource of type `T` is stored.

References to resources

References to global resources returned by `borrow_global` or `borrow_global_mut` mostly behave like references to local storage: they can be extended, read, and written using ordinary reference operators and passed as arguments to other function. However, there is one important difference between local and global references: **a function cannot return a reference that points into global storage**. For example, these two functions will each fail to compile:

```
struct R has key { f: u64 }
// will not compile
fun ret_direct_resource_ref_bad(a: address): &R {
    borrow_global<R>(a) // error!
}
// also will not compile
fun ret_resource_field_ref_bad(a: address): &u64 {
    &borrow_global<R>(a).f // error!
}
```

Move must enforce this restriction to guarantee absence of dangling references to global storage. This section contains much more detail for the interested reader.

Global storage operators with generics

Global storage operations can be applied to generic resources with both instantiated and uninstantiated generic type parameters:

```
struct Container<T> has key { t: T }

// Publish a Container storing a type T of the caller's choosing
fun publish_generic_container<T>(account: &signer, t: T) {
    move_to<Container<T>>(account, Container { t })
}

/// Publish a container storing a u64
fun publish_instantiated_generic_container(account: &signer, t: u64) {
    move_to<Container<u64>>(account, Container { t })
}
```

The ability to index into global storage via a type parameter chosen at runtime is a powerful Move feature known as *storage polymorphism*. For more on the design patterns enabled by this feature, see Move generics.

Example: `Counter`

The simple `Counter` module below exercises each of the five global storage operators. The API exposed by this module allows:

- Anyone to publish a `Counter` resource under their account
- Anyone to check if a `Counter` exists under any address
- Anyone to read or increment the value of a `Counter` resource under any address
- An account that stores a `Counter` resource to reset it to zero
- An account that stores a `Counter` resource to remove and delete it

```
address 0x42 {
module counter {
    use std::signer;

    /// Resource that wraps an integer counter
    struct Counter has key { i: u64 }

    /// Publish a `Counter` resource with value `i` under the given `account`
    public fun publish(account: &signer, i: u64) {
      // "Pack" (create) a Counter resource. This is a privileged operation that
      // can only be done inside the module that declares the `Counter` resource
      move_to(account, Counter { i })
    }

    /// Read the value in the `Counter` resource stored at `addr`
    public fun get_count(addr: address): u64 acquires Counter {
        borrow_global<Counter>(addr).i
    }

    /// Increment the value of `addr`'s `Counter` resource
    public fun increment(addr: address) acquires Counter {
        let c_ref = &mut borrow_global_mut<Counter>(addr).i;
        *c_ref = *c_ref + 1
    }

    /// Reset the value of `account`'s `Counter` to 0
    public fun reset(account: &signer) acquires Counter {
        let c_ref = &mut borrow_global_mut<Counter>(signer::address_of(account)).i;
        *c_ref = 0
    }

    /// Delete the `Counter` resource under `account` and return its value
    public fun delete(account: &signer): u64 acquires Counter {
        // remove the Counter resource
        let c = move_from<Counter>(signer::address_of(account));
        // "Unpack" the `Counter` resource into its fields. This is a
        // privileged operation that can only be done inside the module
        // that declares the `Counter` resource
        let Counter { i } = c;
```

```
        i
    }

    /// Return `true` if `addr` contains a `Counter` resource
    public fun exists(addr: address): bool {
        exists<Counter>(addr)
    }
}
}
```

Annotating functions with `acquires`

In the `counter` example, you might have noticed that the `get_count`, `increment`, `reset`, and `delete` functions are annotated with `acquires Counter`. A Move function `m::f` must be annotated with `acquires T` if and only if:

- The body of `m::f` contains a `move_from<T>`, `borrow_global_mut<T>`, or `borrow_global<T>` instruction, or
- The body of `m::f` invokes a function `m::g` declared in the same module that is annotated with `acquires`

For example, the following function inside `Counter` would need an `acquires` annotation:

```
// Needs `acquires` because `increment` is annotated with `acquires`
fun call_increment(addr: address): u64 acquires Counter {
    counter::increment(addr)
}
```

However, the same function *outside* `Counter` would not need an annotation:

```
address 0x43 {
module m {
   use 0x42::counter;

   // Ok. Only need annotation when resource acquired by callee is declared
   // in the same module
   fun call_increment(addr: address): u64 {
       counter::increment(addr)
   }
}
}
```

If a function touches multiple resources, it needs multiple `acquires`:

```
address 0x42 {
module two_resources {
    struct R1 has key { f: u64 }
    struct R2 has key { g: u64 }

    fun double_acquires(a: address): u64 acquires R1, R2 {
        borrow_global<R1>(a).f + borrow_global<R2>.g
    }
}
}
```

The `acquires` annotation does not take generic type parameters into account:

```
address 0x42 {
module m {
    struct R<T> has key { t: T }

    // `acquires R`, not `acquires R<T>`
    fun acquire_generic_resource<T: store>(a: addr) acquires R {
        let _ = borrow_global<R<T>>(a);
    }

    // `acquires R`, not `acquires R<u64>
    fun acquire_instantiated_generic_resource(a: addr) acquires R {
        let _ = borrow_global<R<u64>>(a);
    }
}
}
```

Finally: redundant `acquires` are not allowed. Adding this function inside `Counter` will result in a compilation error:

```
// This code will not compile because the body of the function does not use a global
// storage instruction or invoke a function with `acquires`
fun redundant_acquires_bad() acquires Counter {}
```

For more information on `acquires`, see Move functions.

Reference Safety For Global Resources

Move prohibits returning global references and requires the `acquires` annotation to prevent dangling references. This allows Move to live up to its promise of static reference safety (i.e., no dangling references, no `null` or `nil` dereferences) for all reference types.

This example illustrates how the Move type system uses `acquires` to prevent a dangling reference:

```
address 0x42 {
module dangling {
    struct T has key { f: u64 }

    fun borrow_then_remove_bad(a: address) acquires T {
        let t_ref: &mut T = borrow_global_mut<T>(a);
        let t = remove_t(a); // type system complains here
        // t_ref now dangling!
        let uh_oh = *&t_ref.f
    }

    fun remove_t(a: address): T acquires T {
        move_from<T>(a)
    }

}
}
```

In this code, line 6 acquires a reference to the `T` stored at address `a` in global storage. The callee `remove_t` then removes the value, which makes `t_ref` a dangling reference.

Fortunately, this cannot happen because the type system will reject this program. The `acquires` annotation on `remove_t` lets the type system know that line 7 is dangerous, without having to recheck or introspect the body of `remove_t` separately!

The restriction on returning global references prevents a similar, but even more insidious problem:

```
address 0x42 {
module m1 {
    struct T has key {}

    public fun ret_t_ref(a: address): &T acquires T {
        borrow_global<T>(a) // error! type system complains here
    }

    public fun remove_t(a: address) acquires T {
        let T {} = move_from<T>(a);
    }
}

module m2 {
    fun borrow_then_remove_bad(a: address) {
        let t_ref = m1::ret_t_ref(a);
        let t = m1::remove_t(a); // t_ref now dangling!
    }
}
}
```

Line 16 acquires a reference to a global resource `m1::T`, then line 17 removes that same resource, which makes `t_ref` dangle. In this case, `acquires` annotations do not help us because the `borrow_then_remove_bad` function is outside of the `m1` module that declares `T` (recall that `acquires` annotations can only be used for resources declared in the current module). Instead, the type system avoids this problem by preventing the return of a global reference at line 6.

Fancier type systems that would allow returning global references without sacrificing reference safety are possible, and we may consider them in future iterations of Move. We chose the current design because it strikes a good balance between expressivity, annotation burden, and type system complexity.

Standard Library

The Move standard library exposes interfaces that implement the following functionality:

- Basic operations on vectors.
- Option types and operations on `Option` types.
- A common error encoding code interface for abort codes.
- 32-bit precision fixed-point numbers.

vector

The `vector` module defines a number of operations over the primitive `vector` type. The module is published under the named address `Std` and consists of a number of native functions, as well as functions defined in Move. The API for this module is as follows.

Functions

Create an empty `vector`. The `Element` type can be both a `resource` or `copyable` type.

```
    native public fun empty<Element>(): vector<Element>;
```

Create a vector of length `1` containing the passed in `element`.

```
    public fun singleton<Element>(e: Element): vector<Element>;
```

Destroy (deallocate) the vector `v`. Will abort if `v` is non-empty. *Note*: The emptiness restriction is due to the fact that `Element` can be a resource type, and destruction of a non-empty vector would violate resource conservation.

```
    native public fun destroy_empty<Element>(v: vector<Element>);
```

Acquire an immutable reference to the `i`th element of the vector `v`. Will abort if the index `i` is out of bounds for the vector `v`.

```
    native public fun borrow<Element>(v: &vector<Element>, i: u64): &Element;
```

Acquire a mutable reference to the `i`th element of the vector `v`. Will abort if the index `i` is out of bounds for the vector `v`.

```
    native public fun borrow_mut<Element>(v: &mut vector<Element>, i: u64): &mut
Element;
```

Empty and destroy the `other` vector, and push each of the elements in the `other` vector onto the `lhs` vector in the same order as they occurred in `other`.

```
    public fun append<Element>(lhs: &mut vector<Element>, other: vector<Element>);
```

Push an element `e` of type `Element` onto the end of the vector `v`. May trigger a resizing of the underlying vector's memory.

```
    native public fun push_back<Element>(v: &mut vector<Element>, e: Element);
```

Pop an element from the end of the vector `v` in-place and return the owned value. Will abort if `v` is empty.

```
    native public fun pop_back<Element>(v: &mut vector<Element>): Element;
```

Remove the element at index `i` in the vector `v` and return the owned value that was previously stored at `i` in `v`. All elements occurring at indices greater than `i` will be shifted down by 1. Will abort if `i` is out of bounds for `v`.

```
    public fun remove<Element>(v: &mut vector<Element>, i: u64): Element;
```

Swap the `i`th element of the vector `v` with the last element and then pop this element off of the back of the vector and return the owned value that was previously stored at index `i`. This operation is O(1), but does not preserve ordering of elements in the vector. Aborts if the index `i` is out of bounds for the vector `v`.

```
    public fun swap_remove<Element>(v: &mut vector<Element>, i: u64): Element;
```

Swap the elements at the `i`'th and `j`'th indices in the vector `v`. Will abort if either of `i` or `j` are out of bounds for `v`.

```
    native public fun swap<Element>(v: &mut vector<Element>, i: u64, j: u64);
```

Reverse the order of the elements in the vector `v` in-place.

```
    public fun reverse<Element>(v: &mut vector<Element>);
```

Return the index of the first occurrence of an element in `v` that is equal to `e`. Returns `(true, index)` if such an element was found, and `(false, 0)` otherwise.

```
    public fun index_of<Element>(v: &vector<Element>, e: &Element): (bool, u64);
```

Return if an element equal to `e` exists in the vector `v`.

```
    public fun contains<Element>(v: &vector<Element>, e: &Element): bool;
```

Return the length of a `vector`.

```
    native public fun length<Element>(v: &vector<Element>): u64;
```

Return whether the vector `v` is empty.

```
    public fun is_empty<Element>(v: &vector<Element>): bool;
```

---

option

The `option` module defines a generic option type `Option<T>` that represents a value of type `T` that may, or may not, be present. It is published under the named address `Std`.

The Move option type is internally represented as a singleton vector, and may contain a value of `resource` or `copyable` kind. If you are familiar with option types in other languages, the Move `Option` behaves similarly to those with a couple notable exceptions since the option can contain a

value of kind `resource`. Particularly, certain operations such as `get_with_default` and `destroy_with_default` require that the element type `T` be of `copyable` kind.

The API for the `option` module is as as follows

Types

Generic type abstraction of a value that may, or may not, be present. Can contain a value of either `resource` or `copyable` kind.

```
    struct Option<T>;
```

Functions

Create an empty `Option` of that can contain a value of `Element` type.

```
    public fun none<Element>(): Option<Element>;
```

Create a non-empty `Option` type containing a value `e` of type `Element`.

```
    public fun some<Element>(e: T): Option<Element>;
```

Return an immutable reference to the value inside the option `opt_elem` Will abort if `opt_elem` does not contain a value.

```
    public fun borrow<Element>(opt_elem: &Option<Element>): &Element;
```

Return a reference to the value inside `opt_elem` if it contains one. If `opt_elem` does not contain a value the passed in `default_ref` reference will be returned. Does not abort.

```
    public fun borrow_with_default<Element>(opt_elem: &Option<Element>, default_ref:
&Element): &Element;
```

Return a mutable reference to the value inside `opt_elem`. Will abort if `opt_elem` does not contain a value.

```
    public fun borrow_mut<Element>(opt_elem: &mut Option<Element>): &mut Element;
```

Convert an option value that contains a value to one that is empty in-place by removing and returning the value stored inside `opt_elem`. Will abort if `opt_elem` does not contain a value.

```
    public fun extract<Element>(opt_elem: &mut Option<Element>): Element;
```

Return the value contained inside the option `opt_elem` if it contains one. Will return the passed in `default` value if `opt_elem` does not contain a value. The `Element` type that the `Option` type is instantiated with must be of `copyable` kind in order for this function to be callable.

```
    public fun get_with_default<Element: copyable>(opt_elem: &Option<Element>, default:
Element): Element;
```

Convert an empty option `opt_elem` to an option value that contains the value `e`. Will abort if `opt_elem` already contains a value.

```
    public fun fill<Element>(opt_elem: &mut Option<Element>, e: Element);
```

Swap the value currently contained in `opt_elem` with `new_elem` and return the previously contained value. Will abort if `opt_elem` does not contain a value.

```
    public fun swap<Element>(opt_elem: &mut Option<Element>, e: Element): Element;
```

Return true if `opt_elem` contains a value equal to the value of `e_ref`. Otherwise, `false` will be returned.

```
    public fun contains<Element>(opt_elem: &Option<Element>, e_ref: &Element): bool;
```

Return `true` if `opt_elem` does not contain a value.

```
    public fun is_none<Element>(opt_elem: &Option<Element>): bool;
```

Return `true` if `opt_elem` contains a value.

```
    public fun is_some<Element>(opt_elem: &Option<Element>): bool;
```

Unpack `opt_elem` and return the value that it contained. Will abort if `opt_elem` does not contain a value.

```
    public fun destroy_some<Element>(opt_elem: Option<Element>): Element;
```

Destroys the `opt_elem` value passed in. If `opt_elem` contained a value it will be returned otherwise, the passed in `default` value will be returned.

```
    public fun destroy_with_default<Element: copyable>(opt_elem: Option<Element>,
default: Element): Element;
```

Destroys the `opt_elem` value passed in, `opt_elem` must be empty and not contain a value. Will abort if `opt_elem` contains a value.

```
    public fun destroy_none<Element>(opt_elem: Option<Element>);
```

errors

Recall that each abort code in Move is represented as an unsigned 64-bit integer. The `errors` module defines a common interface that can be used to "tag" each of these abort codes so that they can represent both the error **category** along with an error **reason**.

Error categories are declared as constants in the `errors` module and are globally unique with respect to this module. Error reasons on the other hand are module-specific error codes, and can provide greater detail (perhaps, even a particular *reason*) about the specific error condition. This representation of a category and reason for each error code is done by dividing the abort code into two sections.

The lower 8 bits of the abort code hold the *error category*. The remaining 56 bits of the abort code hold the *error reason*. The reason should be a unique number relative to the module which raised the error and can be used to obtain more information about the error at hand. It should mostly be used for diagnostic purposes as error reasons may change over time if the module is updated.

Category	Reason
8 bits	56 bits

Since error categories are globally stable, these present the most stable API and should in general be what is used by clients to determine the messages they may present to users (whereas the reason is useful for diagnostic purposes). There are public functions in the `errors` module for creating an abort code of each error category with a specific `reason` number (represented as a `u64`).

Constants

The system is in a state where the performed operation is not allowed.

```
    const INVALID_STATE: u8 = 1;
```

A specific account address was required to perform an operation, but a different address from what was expected was encounterd.

```
    const REQUIRES_ADDRESS: u8 = 2;
```

An account did not have the expected role for this operation. Useful for Role Based Access Control (RBAC) error conditions.

```
    const REQUIRES_ROLE: u8 = 3;
```

An account did not not have a required capability. Useful for RBAC error conditions.

```
    const REQUIRES_CAPABILITY: u8 = 4;
```

A resource was expected, but did not exist under an address.

```
    const NOT_PUBLISHED: u8 = 5;
```

Attempted to publish a resource under an address where one was already published.

```
    const ALREADY_PUBLISHED: u8 = 6;
```

An argument provided for an operation was invalid.

```
    const INVALID_ARGUMENT: u8 = 7;
```

A limit on a value was exceeded.

```
    const LIMIT_EXCEEDED: u8 = 8;
```

An internal error (bug) has occurred.

```
    const INTERNAL: u8 = 10;
```

A custom error category for extension points.

```
    const CUSTOM: u8 = 255;
```

Functions

Should be used in the case where invalid (global) state is encountered. Constructs an abort code with specified `reason` and category `INVALID_STATE`. Will abort if `reason` does not fit in 56 bits.

```
    public fun invalid_state(reason: u64): u64;
```

Should be used if an account's address does not match a specific address. Constructs an abort code with specified `reason` and category `REQUIRES_ADDRESS`. Will abort if `reason` does not fit in 56 bits.

```
    public fun requires_address(reason: u64): u64;
```

Should be used if a role did not match a required role when using RBAC. Constructs an abort code with specified `reason` and category `REQUIRES_ROLE`. Will abort if `reason` does not fit in 56 bits.

```
    public fun requires_role(reason: u64): u64;
```

Should be used if an account did not have a required capability when using RBAC. Constructs an abort code with specified `reason` and category `REQUIRES_CAPABILITY`. Should be Will abort if `reason` does not fit in 56 bits.

```
    public fun requires_capability(reason: u64): u64;
```

Should be used if a resource did not exist where one was expected. Constructs an abort code with specified `reason` and category `NOT_PUBLISHED`. Will abort if `reason` does not fit in 56 bits.

```
    public fun not_published(reason: u64): u64;
```

Should be used if a resource already existed where one was about to be published. Constructs an abort code with specified `reason` and category `ALREADY_PUBLISHED`. Will abort if `reason` does not fit in 56 bits.

```
    public fun already_published(reason: u64): u64;
```

Should be used if an invalid argument was passed to a function/operation. Constructs an abort code with specified `reason` and category `INVALID_ARGUMENT`. Will abort if `reason` does not fit in 56 bits.

```
    public fun invalid_argument(reason: u64): u64;
```

Should be used if a limit on a specific value is reached, e.g., subtracting 1 from a value of 0. Constructs an abort code with specified `reason` and category `LIMIT_EXCEEDED`. Will abort if `reason` does not fit in 56 bits.

```
    public fun limit_exceeded(reason: u64): u64;
```

Should be used if an internal error or bug was encountered. Constructs an abort code with specified `reason` and category `INTERNAL`. Will abort if `reason` does not fit in 56 bits.

```
    public fun internal(reason: u64): u64;
```

Used for extension points, should be not used under most circumstances. Constructs an abort code with specified `reason` and category `CUSTOM`. Will abort if `reason` does not fit in 56 bits.

```
    public fun custom(reason: u64): u64;
```

---

fixed_point32

The `fixed_point32` module defines a fixed-point numeric type with 32 integer bits and 32 fractional bits. Internally, this is represented as a `u64` integer wrapped in a struct to make a unique `fixed_point32` type. Since the numeric representation is a binary one, some decimal values may not be exactly representable, but it provides more than 9 decimal digits of precision both before and after the decimal point (18 digits total). For comparison, double precision floating-point has less than 16 decimal digits of precision, so you should be careful about using floating-point to convert these values to decimal.

Types

Represents a fixed-point numeric number with 32 fractional bits.

```
    struct FixedPoint32;
```

Functions

Multiply a u64 integer by a fixed-point number, truncating any fractional part of the product. This will abort if the product overflows.

```
    public fun multiply_u64(val: u64, multiplier: FixedPoint32): u64;
```

Divide a u64 integer by a fixed-point number, truncating any fractional part of the quotient. This will abort if the divisor is zero or if the quotient overflows.

```
    public fun divide_u64(val: u64, divisor: FixedPoint32): u64;
```

Create a fixed-point value from a rational number specified by its numerator and denominator. Calling this function should be preferred for using `fixed_point32::create_from_raw_value` which is also available. This will abort if the denominator is zero. It will also abort if the numerator is nonzero and the ratio is not in the range $2^{-32}\ldots2^{32}-1$. When specifying decimal fractions, be careful about rounding errors: if you round to display N digits after the decimal point, you can use a denominator of 10^N to avoid numbers where the very small imprecision in the binary representation could change the rounding, e.g., 0.0125 will round down to 0.012 instead of up to 0.013.

```
    public fun create_from_rational(numerator: u64, denominator: u64): FixedPoint32;
```

Create a fixedpoint value from a raw `u64` value.

```
    public fun create_from_raw_value(value: u64): FixedPoint32;
```

Returns `true` if the decimal value of `num` is equal to zero.

```
    public fun is_zero(num: FixedPoint32): bool;
```

Accessor for the raw `u64` value. Other less common operations, such as adding or subtracting `FixedPoint32` values, can be done using the raw values directly.

```
    public fun get_raw_value(num: FixedPoint32): u64;
```

Move Coding Conventions

This section lays out some basic coding conventions for Move that the Move team has found helpful. These are only recommendations, and you should feel free to use other formatting guidelines and conventions if you have a preference for them.

Naming

- **Module names**: should be lower snake case, e.g., `fixed_point32`, `vector`.
- **Type names**: should be camel case if they are not a native type, e.g., `Coin`, `RoleId`.
- **Function names**: should be lower snake case, e.g., `destroy_empty`.
- **Constant names**: should be upper camel case and begin with an `E` if they represent error codes (e.g., `EIndexOutOfBounds`) and upper snake case if they represent a non-error value (e.g., `MIN_STAKE`).
-
- **Generic type names**: should be descriptive, or anti-descriptive where appropriate, e.g., `T` or `Element` for the Vector generic type parameter. Most of the time the "main" type in a module should be the same name as the module e.g., `option::Option`, `fixed_point32::FixedPoint32`.
- **Module file names**: should be the same as the module name e.g., `Option.move`.
- **Script file names**: should be lower snake case and should match the name of the “main” function in the script.
- **Mixed file names**: If the file contains multiple modules and/or scripts, the file name should be lower snake case, where the name does not match any particular module/script inside.

Imports

- All module `use` statements should be at the top of the module.
- Functions should be imported and used fully qualified from the module in which they are declared, and not imported at the top level.
- Types should be imported at the top-level. Where there are name clashes, `as` should be used to rename the type locally as appropriate.

For example, if there is a module:

```
module 0x1::foo {
    struct Foo { }
    const CONST_FOO: u64 = 0;
    public fun do_foo(): Foo { Foo{} }
    ...
}
```

this would be imported and used as:

```
module 0x1::bar {
    use 0x1::foo::{Self, Foo};

    public fun do_bar(x: u64): Foo {
        if (x == 10) {
            foo::do_foo()
        } else {
            abort 0
        }
    }
    ...
}
```

And, if there is a local name-clash when importing two modules:

```
module other_foo {
    struct Foo {}
    ...
}

module 0x1::importer {
    use 0x1::other_foo::Foo as OtherFoo;
    use 0x1::foo::Foo;
    ...
}
```

Comments

- Each module, struct, and public function declaration should be commented.
- Move has doc comments `///`, regular single-line comments `//`, block comments `/* */`, and block doc comments `/** */`.

Formatting

The Move team plans to write an autoformatter to enforce formatting conventions. However, in the meantime:

- Four space indentation should be used except for `script` and `address` blocks whose contents should not be indented.
- Lines should be broken if they are longer than 100 characters.
- Structs and constants should be declared before all functions in a module.