The Move Book

Functions are the building blocks of Move programs. They are called from user transactions and from other functions and group executable code into reusable units. Functions can take arguments and return a value. They are declared with the fun keyword at the module level. Just like any other module member, by default they're private and can only be accessed from within the module.

In this example, we define a function add that takes two arguments of type u64 and returns their sum. The test_add function, located in the same module, is a test function that calls add . The test uses the assert! macro to compare the result of add with the expected value. If the condition inside assert! evaluates to false, the execution is aborted automatically.

In Move, functions are typically named using the snake_case convention. This means function names should be all lowercase, with words separated by underscores. Examples include do_something , add , get_balance , is_authorized , and so on.

A function is declared with the fun keyword followed by the function name (a valid Move identifier), a list of arguments in parentheses, and a return type. The function body is a block of code that contains a sequence of statements and expressions. The last expression the function body is the return value of the function.

Just like other module members, functions can be imported and accessed using a path. The path consists of the module path and the function name, separated by ::. For example, if you have a function named add in the math module within the book package, its full path would be book::math::add . If the module has already been imported, you can access it directly as math::add as in the following example:

Move functions can return multiple values, which is particularly useful when you need to return more than one piece of data from a function. The return type is specified as a tuple of types, and the return value is provided as a tuple of expressions:

The result of a function call with a tuple return has to be unpacked into variables via the let (tuple) syntax:

If any of the declared values need to be declared as mutable, the mut keyword is placed before the variable name:

If some of the arguments are not used, they can be ignored with the _ symbol:

Function declaration

In Move, functions are typically named using the snake_case convention. This means function names should be all lowercase, with words separated by underscores. Examples include do_something , add , get_balance , is_authorized , and so on.

A function is declared with the fun keyword followed by the function name (a valid Move identifier), a list of arguments in parentheses, and a return type. The function body is a block of code that contains a sequence of statements and expressions. The last expression the function body is the return value of the function.

```
bash fun return_nothing() { // empty expression, function returns `()` }
```

Just like other module members, functions can be imported and accessed using a path. The path consists of the module path and the function name, separated by ::. For example, if you have a function named add in the math module within the book package, its full path would be book::math::add . If the module has already been imported, you can access it directly as math::add as in the following example:

```bash module book::use_math;

use book::math;

fun call_add() { // function is called via the path let sum = math::add(1, 2); } ```
```

Move functions can return multiple values, which is particularly useful when you need to return more than one piece of data from a function. The return type is specified as a tuple of types, and the return value is provided as a tuple of expressions:

```
bash fun get_name_and_age(): (vector<u8>, u8) { (b"John", 25) }
```

The result of a function call with a tuple return has to be unpacked into variables via the let (tuple) syntax:

```
bash // Tuple must be destructured to access its elements. // Name and age are declared as
immutable variables. let (name, age) = get_name_and_age(); assert!(name == b"John"); assert!(age ==
25);
```

If any of the declared values need to be declared as mutable, the mut keyword is placed before the variable name:

```
bash // declare name as mutable, age as immutable let (mut name, age) = get_name_and_age();
```

If some of the arguments are not used, they can be ignored with the _ symbol:

```
bash // ignore the name, only use the age let (_, age) = get_name_and_age();
```

Accessing functions

Just like other module members, functions can be imported and accessed using a path. The path consists of the module path and the function name, separated by ::. For example, if you have a function named add in the math module within the book package, its full path would be book::math::add . If the module has already been imported, you can access it directly as math::add as in the following example:

```bash module book::use_math;

use book::math;

fun call_add() { // function is called via the path let sum = math::add(1, 2); } ```
```

Move functions can return multiple values, which is particularly useful when you need to return more than one piece of data from a function. The return type is specified as a tuple of types, and the return value is provided as a tuple of expressions:

```
bash fun get_name_and_age(): (vector<u8>, u8) { (b"John", 25) }
```

The result of a function call with a tuple return has to be unpacked into variables via the let (tuple) syntax:

```
bash // Tuple must be destructured to access its elements. // Name and age are declared as
immutable variables. let (name, age) = get_name_and_age(); assert!(name == b"John"); assert!(age ==
25);
```

If any of the declared values need to be declared as mutable, the mut keyword is placed before the variable name:

```
bash // declare name as mutable, age as immutable let (mut name, age) = get_name_and_age();
```

If some of the arguments are not used, they can be ignored with the _ symbol:

```
bash // ignore the name, only use the age let (_, age) = get_name_and_age();
```

Multiple return values

Move functions can return multiple values, which is particularly useful when you need to return more than one piece of data from a function. The return type is specified as a tuple of types, and the return value is provided as a tuple of expressions:

```
bash fun get_name_and_age(): (vector<u8>, u8) { (b"John", 25) }
```

The result of a function call with a tuple return has to be unpacked into variables via the let (tuple) syntax:

```
bash // Tuple must be destructured to access its elements. // Name and age are declared as
immutable variables. let (name, age) = get_name_and_age(); assert!(name == b"John"); assert!(age ==
25);
```

If any of the declared values need to be declared as mutable, the mut keyword is placed before the variable name:

```
bash // declare name as mutable, age as immutable let (mut name, age) = get_name_and_age();
```

If some of the arguments are not used, they can be ignored with the _ symbol:

```
bash // ignore the name, only use the age let (_, age) = get_name_and_age();
```

Further reading