Functions Tutorial

Defining functions

Functions can be defined using the `function` keyword and are covered in section 2.5.8 [1] of the Reference Manual. There are two variations of the way in which they can be declared:

> **`function`** *function_name* **`(`** *args* **`)`** *body* **`end`**

is the same as:

> *function_name* **`= function(`** *args* **`)`** *body* **`end`**

The following is an example of a simple function to double a number:

```
> function foo(n) return n*2 end
> = foo(7)
14
```

We define a function called "foo" which takes a single argument and returns twice its value. The above function could equally have been written:

```
> foo = function(n) return n*2 end
> = foo(4)
8
```

When we define a function you could say we are assigning a function body to a variable. This ability to treat functions as we would treat any other object which we can assign to a variable means that functions are *first class values*. The variable that has a function assigned to it then has the type *function*. We don't have to specify a complicated function type to the variable, e.g. C function pointers which contain argument and return type information.

Function prototypes

Lua has no need for function prototypes! Lua is a *dynamically typed* language with *first class objects*. This means that we only find out whether something is a function when we try to call it at runtime. *Statically typed* languages (like C) have to know what type everything is at compile time.

```
> x = "onion"
> x()
stdin:1: attempt to call global `x' (a string value)
stack traceback:
        stdin:1: in main chunk
        [C]: ?
```

We could not call the object `x` because it is a string. When we assign the function `foo` to the variable `x`, we can call it (as in the previous example). Notice how we do not have the complication of function prototypes or having to tell Lua it is a function.

```
> x = foo
> = x(77)
154
```

We have no problem assigning functions with different prototypes to the variable `x`.

```
> x = function(a,b) return a+b end
> = x(5,6)
11
> function x(a,b) return a..b end
> = x('a','b')
ab
```

Of course we must be careful that we don't accidently name variables the same as we won't get the same errors reported on compilation, or at runtime as we would with a statically typed language. However, this functionality is very useful and allows us to write very flexible code simply.

Arguments

Since Lua has no prototypes it needs to deal with function arguments and return values in a flexible manner. Lua handles multiple arguments, variable argument lists and multiple return values. This is covered in more detail in the FunctionCallTutorial.

Anonymous functions

It is sometimes useful to define functions to perform tasks without giving them a name by assigning them to a variable. We will insert an function definition into a modification of the above example. We'll define a function to pass to `table.foreach()` which for the variable argument list displays each key, its value and the type of the value object:

```
> function foo(...)
>>  table.foreach(arg, function(key,value)
```

```
>>                      print(key,value,type(value))
>>                    end)
>> end -- foo
> foo("apple",2,"banana",3.1415927,foo)
1       apple   string
2       2       number
3       banana  string
4       3.1415927       number
5       function: 004419E8      function
n       5       number
```

Notice how the function we define has no name:

```
function(key,value)
  print(key,value,type(value))
end
```

This is called an *anonymous function*. The function is defined and instead of being assigned to a variable, a reference to it is passed as an argument to `table.foreach`.

First Class Value Example

In the following example we create a function called "`foo`", which we can see has type "`function`":

```
> function foo()
>>  print("foo!")
>> end
>
> foo()
foo!
> = type(foo)  -- what type is foo?
function
```

Since `foo` is just a reference to a function body we can assign other variables to have the same value. We can also delete the reference to `foo` by assigning the value `nil` to it. This effectively deletes the `foo` variable.

```
> bar = foo    -- copy foo's reference to another variable
> foo = nil    -- delete the reference to the function body
> = foo
nil
> foo()        -- try to invoke the function
```

```
stdin:1: attempt to call global `foo' (a nil value)
stack traceback:
        stdin:1: in main chunk
        [C]: ?
```

Note, if no variables point to the function body that we defined it will be unreachable, and will be deleted (see GarbageCollectionTutorial). In this instance we assigned the function body to the variable `bar`, so in a way we renamed the function. But when we set the variable `bar` to `nil`, there are no other variables pointing to the function body, and it will be deleted when the garbage collector is called.

```
> bar()         -- bar still points to the same function that foo did
foo!
> bar = nil     -- now nothing points to the function so it can be garbage collected
```

Note that we did not have to specify function prototypes, which you can read about above, or have to worry about how arguments will be passed. This is because Lua is a dynamically typed language where a variable can point to an object of any type. The variable is really just a reference to an object and the type comes from the object referenced.

FindPage · RecentChanges · preferences
edit · history
Last edited January 14, 2004 6:14 pm PDT (diff)