Anonymous functions, or *funs*, address that problem by letting you declare a special kind of function inline, without naming them. They can do pretty much everything normal functions can do, except calling themselves recursively (how could they do it if they are anonymous?) Their syntax is:

```erlang
fun(Args1) ->
  Expression1, Exp2, ..., ExpN;
  (Args2) ->
  Expression1, Exp2, ..., ExpN;
  (Args3) ->
  Expression1, Exp2, ..., ExpN
end
```

And can be used the following way:

```erlang
7> Fn = fun() -> a end.
#Fun<erl_eval.20.67289768>
8> Fn().
a
9> hhfuns:map(fun(X) -> X + 1 end, L).
[2,3,4,5,6]
10> hhfuns:map(fun(X) -> X - 1 end, L).
[0,1,2,3,4]
```

And now you're seeing one of the things that make people like functional programming so much: the ability to make abstractions on a very low level of code. Basic concepts such as looping can thus be ignored, letting you focus on what is done rather than how to do it.

Anonymous functions are already pretty dandy for such abstractions but they still have more hidden powers:

```erlang
11> PrepareAlarm = fun(Room) ->
   io:format("Alarm set in ~s.~n",[Room]),
   fun() -> io:format("Alarm tripped in ~s! Call Batman!~n",[Room])
end
11> end.
#Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.6.13229925>
13> AlarmReady().
Alarm tripped in bathroom! Call Batman!
ok
```
Hold the phone Batman! What's going on here? Well, first of all, we declare an anonymous function assigned to `PrepareAlarm`. This function has not run yet: it only gets executed when `PrepareAlarm("bathroom")` is called.

At that point, the call to `io:format/2` is evaluated and the "Alarm set" text is output. The second expression (another anonymous function) is returned to the caller and then assigned to `AlarmReady`. Note that in this function, the variable `Room`'s value is taken from the 'parent' function (`PrepareAlarm`). This is related to a concept called **closures**.

To understand closures, one must first understand scope. A function's scope can be imagined as the place where all the variables and their values are stored. In the function `base(A) -> B = A + 1.`, `A` and `B` are both defined to be part of `base/1`'s scope. This means that anywhere inside `base/1`, you can refer to `A` and `B` and expect a value to be bound to them. And when I say 'anywhere', I ain't kidding, kid; this includes anonymous functions too:

```
base(A) ->
  B = A + 1,
  F = fun() -> A * B end,
  F().
```

`B` and `A` are still bound to `base/1`'s scope, so the function `F` can still access them. This is because `F` inherits `base/1`'s scope. Like most kinds of real-life inheritance, the parents can't get what the children have:

```
base(A) ->
  B = A + 1,
  F = fun() -> C = A * B end,
  F(),
  C.
```

In this version of the function, `B` is still equal to `A + 1` and `F` will still execute fine. However, the variable `C` is only in the scope of the anonymous function in `F`. When `base/1` tries to access `C`'s value on the last line, it only finds an unbound variable. In fact, had you tried to compile this function, the compiler would have thrown a fit. Inheritance only goes one way.

It is important to note that the inherited scope follows the anonymous function wherever it is, even when it is passed to another function:

```
a() ->
  Secret = "pony",
  fun() -> Secret end.

b(F) ->
  "a/0's password is " ++ F().
```

Then if we compile it:
Who told a/0's password? Well, a/0 did. While the anonymous function has a/0's scope when it's declared in there, it can still carry it when executed in b/1, as explained above. This is very useful because it lets us carry around parameters and content out of its original context, where the whole context itself are not needed anymore (exactly like we did with Batman in a previous example).

You're most likely to use anonymous functions to carry state around when you have functions defined that take many arguments, but you have a constant one:

```erl
16> math:pow(5, 2).
25.0
17> Base = 2.
2
18> PowerOfTwo = fun(X) -> math:pow(Base, X) end.
#Fun<erl_eval.6.13229925>
17> hhfuns:map(PowerOfTwo, [1, 2, 3, 4]).
[2.0, 4.0, 8.0, 16.0]
```

By wrapping the call to `math:pow/2` inside an anonymous function with the `Base` variable bound in its scope, we made it possible to have each of the calls to `PowerOfTwo` in `hhfuns:map/2` use the integers from the list as the exponents of our base.

A little trap you might fall into when writing anonymous functions is when you try to redefine the scope:

```erl
base() ->
A = 1,
(fun() -> A = 2 end)().
```

This will declare an anonymous function and then run it. As the anonymous function inherits `base/0`'s scope, trying to use the `=` operator compares 2 with the variable `A` (bound to 1). This is guaranteed to fail. However it is possible to redefine the variable if it's done in the nested function's head:

```erl
base() ->
A = 1,
(fun(A) -> A = 2 end)(2).
```

And this works. If you try to compile it, you'll get a warning about shadowing ("Warning: variable 'A' shadowed in 'fun'"). Shadowing is the term used to describe the act of defining a new variable that has the same name as one that was in the parent scope. This is there to prevent some mistakes (usually rightly so), so you might want to consider renaming your variables in these circumstances.

**Update:**

Starting with version 17.0, the language supports using anonymous functions with an internal name. That's right, **anonymous but named functions**.
The trick is that the name is visible only within the function's scope, not outside of it. The main advantage of this is that it makes it possible to define anonymous recursive functions. For example, we could make an anonymous function that keeps being loud forever:

```
18> f(PrepareAlarm), f(AlarmReady).
ok
19> PrepareAlarm = fun(Room) ->
   io:format("Alarm set in ~s.~n", [Room]),
   fun Loop() ->
     io:format("Alarm tripped in ~s! Call Batman!~n", [Room]),
     timer:sleep(500),
     Loop()
   end
end.
#Fun<erl_eval.6.71889879>
20> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.44.71889879>
21> AlarmReady().
Alarm tripped in bathroom! Call Batman!
Alarm tripped in bathroom! Call Batman!
Alarm tripped in bathroom! Call Batman!
...
```

The `Loop` variable refers to the anonymous function itself, and within that scope, will be usable as any other similar variable pointing to an anonymous function. This should generally make a lot of operations in the shell a lot less painful moving on forward.

We'll set the anonymous function theory aside a bit and we'll explore more common abstractions to avoid having to write more recursive functions, like I promised at the end of the previous chapter.

Source: http://learnyousomeerlang.com/higher-order-functions