We are going to talk about Continuations.

To understand what continuations are we need to have in mind two things: closures and the control flow of our program. If we get this, continuations will be easy.

We are going to talk a bit about control flow, how functions (and closures) are also control flow structures and then we are going to talk about what continuations are and how we can use them.

Finally, I’ll try my best to give real life examples on where continuations can be used. So, if at some point early on you have the urge to say “but can’t this be done easier with X?”, the answer is probably yes. But try to focus on the techniques. And eventually we will get to more interesting examples.

go with the control flow

Control flow is the order in which the statements of your program are executed. Actually, it’s about controlling the order in which the statements of your program are executed.

Let’s think a bit about which control flow mechanisms we know and what they actually do.

At the most basic, the CPU has a control flow policy: every time it reads an instruction to execute, it needs to decide what is the next instruction it’s going to execute (which usually is “the next instruction in memory”). This notion becomes so natural that most (but not all) programming languages have some notion of this idea:

Most language structures are control flow structures: if, while, switch, exceptions, for loops. Since for-loops are forbidden in this class, let’s see a for loop example:
for loops are weird in functional languages. Nobody does it like that. How would we usually do something like this in F#?

fold is a function that reduces a list to a single element. The interesting point here is that we are replacing the for loop with a recursive call to fold. This works because function calls are also a control flow structure. Let’s talk a bit about that.

functions, closures and the pursuit of happiness

What do we know about functions?

We know that in Functional languages they are first class citizens. We know that they have arguments and a body. Sometimes they even have names:

```fsharp
let awesomestFunction f x = f (f x)
awesomestFunction (fun x -> x * x) 3 // ==> 81
```

We know more things about functions. They can be associated with an environment in a closure:
As you can see here `creature` is a function. But every time you call `creator` you get a different closure (on the same function).

There’s one more thing: functions can be called and return values and that’s what makes them a control flow structure: when they are called, they change the execution of your program to the function and after they are done, they change it back to where the function was originally called.

Let’s quickly think about that. When a function is called, it’s pretty trivial where the execution has to go to. But how do functions know where to return to?

The way functions know where to return to is by using *The Stack*. *The Stack* is a stack of “execution frames” or “stack frames”. Those execution frames contains all the runtime information needed to call a function (like arguments) and for the function to return to its callee.

Different languages have different ways of building this execution frame (and some don’t even have that).

Usually when a function in a Stack-based language gets called, here’s what happens: the callee has to *push* an execution frame on The Stack that contains the place where the function has to return to, and then it changes execution to the function. The function, on the other hand, when it’s time for it to return, assumes there’s a properly set up execution frame on The Stack, *pops* it up, and return to the place where that frame points to.

The important bit here is: each stack execution frame must be preserved while the called function is still running so the function knows where to go back to once it’s done.

Let’s see an example:

```plaintext
let creator x =
  let creature y = x * y
  creature

let c1 = creator 1
let c2 = creator 2

// result: 3
result = c1 3
// result: 6
result = c2 3
```
When this code executes, it adds a frame stack to call `do_something` and another one when `do_something` calls `divide_failwith`. When this function ends, it pops up the stack, and then goes back to right after `divide_failwith` was called. The next thing is the end of `do_something`, so it pops another stack and then returns to the main of our program.

When `failwith` is called, it raises an exception, which ends up showing:

```
System.Exception: Divisor cannot be zero.
at divide_failwith (Int32 x, Int32 y)
at do_something (Int32 a, Int32 b)
at main@ ()
Stopped due to error
```

What F# does when it realizes there was an uncaught exception is to “dump the stack”, i.e., it goes through each frame in The Stack and prints which function it belongs to. The objective is to try to help you debug the error. It can do that since this information is already available, because it’s part of the way the function calling control flow is done.

So The Stack is great, right? It gives us the proper function calling and free debugging info. Awesome. So what’s the problem with The Stack?

The problem with The Stack is that it has limited space available. Each stack frame takes space which means there’s a limit on how many function you can call inside functions.

In functional languages, where recursion is often used, it becomes a big deal. All of the sudden there’s a hard limit on how much recursion you can do, which is not great.

A solution to that is the so called tail call elimination. It works by the compiler realize that, if a function always ends by calling some other function it doesn’t need to create a stack frame for it. Instead it can just jump to the new function and the stack frame used for the initial function will just be reused by the new one.
Continuations through the looking glass

So, now we know a few things: that all languages provide ways for us to change the control flow of our program (that’s what makes computer programs interesting); and that functions are control flow mechanisms that use a stack to jump around.

We are going to see a new control flow mechanism. It’s called Continuations.

Like functions, continuations change the order things get executed on a program and can be on a closure. But unlike functions, they don’t depend on the growth of the stack. They allow you to build a program where the control flow can be directly manipulated.

So, what continuations look like and how can we use them? I’m glad you asked this, Emma. Fortunately for us, all languages that have first class functions and tail-call elimination already allow for continuations. In those languages, like F#, continuations are simply closed functions that are not expected to return.

Think about it: if a function is not supposed to return, then there’s no need for a stack (Which tail-call elimination will realize and get rid of). So we get the benefits of a function without the downside.

But if that was all, it would be a bit dull, no? A function that just doesn’t return it’s like a dead end. And also, doesn’t a function always return when it finishes?

You are right, Emma. The missing part is: instead of returning, continuations are expected to call other functions when they are done.

Let’s see some code:

```plaintext
1 let print_number x = 
2  printf "%.0f\n" x 
3  System.Environment.Exit 0
4
5 let add_and_print x = 
6  let y = 17.0 + x 
7  print_number y
8
9 let square_add_and_print x = 
10  let y = x * x 
11  add_and_print y
12
13 square_add_and_print 5.0 // prints 42 and exits.
```

First, notice that all 3 functions print_number, add17_and_print and
square_add_and_print are continuations: they are functions with closures that are not expected to return and instead call other continuations to continue the flow of the program. Interestingly, Exit is also a function that is not expected to return.

But who would program like this? Well, for something simple, nobody. That’s why it’s called an example, Greg.

How can we make this more interesting? Well, we realized that continuations must call other continuations when they are done, right? What if, instead of hard coding the next continuation (as we did on the previous examples), we pass it as a parameter?

```plaintext
let print_number x = 
 printf "%.0f\n" x
System.Environment.Exit 0

let calculate x cont = 
 let y = x * x + 17.0 
 cont y

calculate 5.0 print_number // prints 42 and exits.
```

Now we are getting somewhere! We have two functions that each do something meaningful and we connect them by telling the first function: “when you are done, this is where you are supposed to go”. We even got to reuse print_number from the previous example.

Look at what we just did. We were able to construct a function continuation calculate that has a semantic of “give me a value, I’ll do some computation with it and when I’m done I’ll send it to the function you pass me”.

This is normally called “Continuation Passing Style” or CPS. I.e., we are programming in a way that we always pass the continuations for our program. We are going to focus on this for most of this class, and later I’ll get back to continuations in general.

**OMGBBBQ Continuations**

Ok. Let’s do one more example. Imagine we want to write a simple login scheme using CPS: ask for a username, password, check if everything is ok and login the user.
Pay a bit of attention to `do_login`: it’s a *continuation*, i.e., a function that never returns. But the caller of `do_login` can still control what happens after the login, by telling it to call `greet_user` when everything is finished.

Also, it’s important that you think of each of the functions in isolation when we are thinking in terms of continuations. In a way, it’s very similar to how we think about recursion (if we don’t want to be confused by it). Each function has a clear mandate and responsibility: it gets some params, does what it needs to do and calls the continuation it promised to call. It’s almost an afterthought that everything works together as we want, as long as each block does what it said it was going to do.

Moving on, our `check_password` is pretty lame, it just continues. Imagine that we want to do some real work with it. In that case, maybe having that single continuation is kinda silly, since you probably don’t want to do the same thing when the login fails or succeeds.

How could we solve this? Well, since we are passing the continuation as a parameter, maybe we can pass two continuations: one for when the password checks, and one for when it doesn’t.
Let’s look at `check_password`. It continues to `valid_cont` if the password is valid, or to `fail_cont` if it isn’t. It’s a simple `if` statement, but there’s something subtle going on here: we are using our continuations as mechanisms to change the flow of the program.

The tricky part here is on line 17, when we define the `when_fail` continuation. Let’s think this from `do_login`’s perspective: it wants to call `check_password` and move forward with the process (with `login_cont`). But now it has to deal with the password being wrong. What does it want? One reasonable option is: if the password check fails, let’s try the login again. How do we do this? We pass it a continuation that calls `do_login` again. But what is the continuation for this new `do_login` call? Well, the same as before, `login_cont`.

Don’t just “I get it” this. Think a little bit further. We are constructing a new continuation on the fly to do what we want. In this case, it’s nothing super interesting: just doing the same thing the caller did when calling `do_login`. But we are still building the sequence of code that will happen “on the fly”.

```
let check_password user pass valid_cont fail_cont =
  if pass = "1729" then
    printfn "Password is OK."
    valid_cont user
  else
    printfn "Wrong password."
    fail_cont ()

let rec do_login login_cont =
  printf "login: 
  let u = System.Console.ReadLine()
  printf "password: 
  let p = System.Console.ReadLine()
  let when_fail = fun () -> do_login login_cont
  check_password u p login_cont when_fail

let greet_user user =
  printfn "Hey, %s, welcome!" user
  finish ()

do_login greet_user
// login: drumpf
// password: 1
// Wrong password.
// login: drumpf
// password: 1729
// Password is OK.
// Hey, drumpf, welcome!
```
We are getting good at this

Ok, let’s see another example. Imagine we want to write a function that sums all numbers from 0 to n.

```ml
let rec stack_sum n =
  if n = 0 then 0
  else n + stack_sum (n - 1)
```

This is not a tail-call. Can you see why? The last function called is not `stack_sum`, but `+` (plus)!

What happens when we run this?

```ml
stack_sum 1000  // ==> 500500
stack_sum 1000000  // ==> None
// Process is terminated due to StackOverflowException.
```

But remember how we realized that all continuations are tail-call optimizable and therefore can’t stack overflow? So if we rewrite our function using continuations, we won’t have this problem. Let’s try:

```ml
let rec cont_sum n cont =
  if n = 0 then cont 0
  else cont_sum (n - 1) (fun x -> cont (x + n))
cont_sum 1000 print_number  // prints 500500
```

Let’s read this slowly. First, what is our function signature? `cont_sum` receives a `n` that we want to count up to (from 0 to n) and a continuation `cont` to be called with the result. This continuation could be `print_number` like we had before.

The `then` statement is pretty obvious, right? If we have to sum to zero, we call our continuation with the result 0.

The `else` statement is the good one. We are defining `cont_sum n` as being equivalent to `cont_sum (n-1)` with a different continuation. What is this continuation? It’s the previous continuation wrapped with `+n`.

Let’s say this in another way: `cont_sum` keeps building up a continuation: the original one `+ n + n-1 + n-2 + ...` until we get to the 0 case, when we call that bigger continuation with 0, which then calculates all the sum and call the original continuation.
This is an important step and just like last time, let’s recap what we are doing: we are also creating a continuation on the fly, but this continuation is not as dull as last time. It’s doing the actual heavy work of our function. The recursion is used to actually build a continuation, that eventually gets executed all in one shot.

Our new function doesn’t stack overflow, so:

```
1 cont_sum 1000000 print_number  // prints 500000500000
```

oh yeah.

One very quick warning to people doing F# on Mac and Linux. On those platforms, we run our F# code using Mono. Unfortunately, there is an open bug on Mono, because it does not support tail calls at all (i.e. tail calls still increase the stack). You can see the tail call being optimized if you check the compiled code (with the `monodis` tool), but the example above still fails.

Another extra note: sometimes people rename the continuation variable `return`, so the code looks pretty much like a regular function where you `return` the value. It may make for a good gimmicky if you are really confused. But it’s very misleading, since the thing that’s usually associated with `return` (i.e., the actually go back to where you came from) is definitely not happening.

Exercise: can you implement factorial and fibonacci using continuations?

---

**Say continuation one more time, I dare you**

Let’s keep working on examples of using CPS to solve interesting problems. Let’s imagine we have a binary tree structure:
Now let's write a function that returns true if an element is on this tree, using a standard recursive approach:

```ocaml
let rec stack_find el t = match t with
| Empty -> false
| (x, l, r) ->
  if x = el
  then true
  else if stack_find el l
  then true
  else stack_find el r

stack_find 'g' treel // == true
stack_find 'h' treel // == false
```

We recursively look at the current element, the left branch and then the right branch, returning if we ever find the element. But look at what happens at runtime:

```ocaml
stack_find 'g' treel = find (A)
  -> find (C)
  -> find (F)
    -> find (G)
      -> true
```

We keep recursing down the tree and building the stack. As soon as we find the answer, we return true. But this true has to propagate back all the way up the stack. Each function
returns `true`, then goes back to its callee, that then returns `true`, etc, etc...

Gladly, we know how to do better now. Ideally we want to return as soon as we found something, and not worry about whatever happened before. Let’s re-implement this using continuations. Instead of returning `true` or `false` we can have a continuation for success (the element exists) and another one for fail (the element doesn’t exist):

```ocaml
let rec cont_find el t succ fail = match t with
  | Empty -> fail ()
  | Branch (x, l, r) ->
    if x = el
      then succ ()
      else cont_find el l succ (fun () -> cont_find el r succ fail)

let yes () = printfn "yes"
let no () = printfn "no"

cont_find 'g' treel yes no // prints yes
cont_find 'h' treel yes no // prints no
```

Wow. Just wow. Isn’t this beautiful?

Ok, let’s go over what’s happening. First, let’s check our function signature. `cont_find` receives 4 parameters: `el` and `t` are the same as before (the element we are looking for and the tree we are looking into). The other two are continuations: `succ` is supposed to be called when we find the element, `fail` when we don’t.

So what does the function do? If we are in an empty branch, there’s obviously nowhere else to go, so we continue to `fail` (i.e., we didn’t find the element). If we are in a branch and we have the node we were looking for, we call `succ` (i.e., we found the element). Fair.

Now what happens when we don’t have the element and we have the left and right branches to check: we call `cont_find` on the left branch. What are the continuations for this call? If we succeed we just want to call the `succ` we had before. But if we fail, we still have to check the right branch. So we do this. And what are the continuations for this new `fail`? Well, for success, is always the same. And there’s nothing else we want to do, so failing here is the same as failing the whole thing, so we just call the original `fail`.

Another way of putting this: we first check the left side of our tree, but as we go down we build a bigger `fail` continuation. this fail continuation contains the find for the right side of the tree. We are saying: Try to find the element here, but if you fail you still need to look on this other side before really giving up (i.e., before calling the `fail` the original callee gave us).

Exercise: can you change `cont_find` to use a single continuation that receives a boolean
telling if the element was found or not?

I like continuations so much I want another example

maybe this should be another exercise?

One last example. Let’s write a function `cut` that given a list returns the sublist up to the point where a passed function is true. For example:

```plaintext
let isEven x = (x % 2) = 0

let cut list pred = cut [ 2 ; 4 ; 6 ; 5 ; 2 ] isEven // ==> [ 2 ; 4 ; 6 ]
let cut [ 1 ; 3 ; 5 ] isEven // ==> []
```

notice that `cut` is not CPS. But we are going to use CPS inside it to solve the problem.

```plaintext
let cut list pred = 
  // we first create our internal CPS function.
  let rec cc list pred cont = 
    match list with 
    | [] -> cont []
    | x::xs -> 
      if (pred x) 
        then cc xs pred (fun a -> cont (x :: a))
      else cont []
    // then we call it. Notice that our continuation is 
    // the identity function:
    // let id = fun x -> x
    cc list pred id
```

It’s similar to our previous examples: we change the continuation for the recursive call to be the original one wrapped with `x ::`. When we finally get a value that doesn’t pass `pred`, we immediately continue.

(I can’t get no) exception

We are going to talk now about 2 common uses of continuations “in the wild”. The first one is how to do exceptions with continuations. It’s actually pretty similar to the list find example we just saw.

If we think about how exceptions work (they are also a control flow structure), we have the
following:

1. a statement can thrown an exception.
2. after each statement there are two possible paths our code can take: continue with the normal flow of the program, or go to the exception flow.
3. the exception flow can also go both ways: return to the normal flow (if it caught that particular exception) or stop everything we’ve been doing (since the exception hasn’t been caught).

Too abstract? Let’s see what an exception looks like:

```plaintext
let ret =
try
  failwith "fail"
  "hello"
with
  | Failure msg -> "caught: " + msg
ret  // ==> "caught: fail"
```

So here `ret` becomes “caught: fail”, since `failwith` is caught by our `try/with` statement. Interestingly, in many languages exceptions are actually implemented as continuations. The reason being that exceptions can have extremely complicated interactions with other control flow structures - think of a `break` inside of a `for` inside of an exception handler. Therefore it’s not uncommon for compilers to transform your code into a CPS-style code during compilation to solve this (we will see something like this later on).

So, let’s do the other way around. Let’s think how this would be implemented using continuations. Well, lucky for us, our previous example looks very similar to what we want: we want to write some code that represents the “normal” flow of our program, and some code for when something goes wrong. On our `cont_find` function, we called it `succ` and `fail` and we didn’t think of them as actually something going wrong. It was just two options on our code. We can instead phrase them as a regular continuation and an error one.

Remember the `divide_failwith` example we had before:

```plaintext
let divide_failwith x y =
  if y = 0 then failwith "Divisor cannot be zero."
  else
    x / y
divide_failwith 4 2  // ==> 2
```

Rewriting it with CPS replacing the exception would look like:
```plaintext
let divide_failwith x y cont error =
  if y = 0 then error "Divisor cannot be zero."
  else
    cont (x / y)
```

It’s that simple. To call this, we need some code to run as an exception handler. We could have a `default_error` continuation to use everywhere:

```plaintext
let default_error msg =
  printfn "Uncaught exception: %s" msg
  System.Environment.Exit 1
```

And then we can use:

```plaintext
divide_failwith 4.0 2.0 print_number default_error // 2.0
divide_failwith 4.0 0.0 print_number default_error
  // Uncaught exception: Divisor cannot be zero.
```

We can also implement `finally`, by simply extending both continuations with our code. Imagine we want to call a function `close_database` in our `finally` statement (i.e., we want that function to be called on all possible outcomes of our code):

```plaintext
// we must call close_database. So instead of:
// divide_failwith x y print_number default_error
// we have:
let cont = fun x ->
  close_database ()
  print_number x

let fail = fun x ->
  close_database ()
  default_error x

divide_failwith 4.0 2.0 cont fail
```

In this case `close_database` behaves as a `finally` statement, i.e., it gets called when `divide_failwith` returns, no matter if an error happened or not.

We could go further with our exception mechanism and, for example, pass the `cont` continuation to the exception one, allowing it to recover from an error. But the important thing is the principle of what we are doing: making the control flow of our program explicit and implementing features that may not even be directly available in our code’s language (as long as it supports continuations).
So call(back) me maybe

Our second generic application of continuations are callbacks. Callback are interesting because they are probably the simplest form of continuations and yet they are so useful that they are used everywhere in modern “real life” programming.

First, what are callbacks? Callbacks are continuations that we pass in otherwise non-continuation style programs to continue a computation after a certain event has occurred.

For example, imagine that we want to load a file from a URL on the Web. It may take quite a while for the whole file to be downloaded (and it may even fail). So we don’t want our program to hang there and wait for the file to complete. In this case, we want an API that allows us to say: here’s a URL I want, and please call this continuation when you are done downloading it and ready for me to use the file. In theory, it’s identical to the type of code we’ve been writing here. In practice, the only difference is that our continuation will be called sometime in the future (not right away).

For illustration purposes, here is actual code from Google Chrome:

```c
1     void OnThumbnailAvailable(RequestContext* context,
2            const GURL& url,
3            const SkBitmap* bitmap) {
4       if (bitmap) {
5           // ... use bitmap as thumbnail of the page.
6       }
7     }
8     thumbnail_manager_-->GetImageForURL(url, &OnThumbnailAvailable);
```

We can see that thumbnail_manager_ provides an API that gets an URL and a continuation for when the image has been loaded. In this case OnThumbnailAvailable is our callback (continuation).

Callbacks are also extremely popular in JavaScript (that runs on all web pages), since most events (a mouse click, a keyboard press, a form submit) are handled this way. So it’s super common in Javascript to see code like:

```javascript
1   function onKeyPress(event) {
2       // some key has just been pressed.
3   }
4   document.addEventListener("keypress", onKeyPress);
```

Again, what the API asks with the callback is a continuation (onKeyPress in this case): when
this event happen (a key press), stop whatever you are doing and continue the program execution there.

Notice that I’m not showing you any code in F# with callbacks. The reason is: callbacks are such a simpler use case of continuations, that all our previous examples are much more interesting than those. Still, I want you to remember that we are doing an advanced version of something that is used everywhere on day-to-day programming.

**CPS Transformers: the compiler**

Remember when we talked about how exceptions can (and often are) written with continuations? It so happens that a lot of functional languages (Haskell, for example) have an intermediate representation in Continuation-Passing Style during compilation (as opposed to static single assignment style - SSA - for imperative languages).

Luckily, it is relatively simple to automatically (or manually, for that matter) transform any non-CPS code into CPS code. I won’t go into compiler theory here, but I want us to have a feeling of how this work. Imagine those 4 bindings:

```plaintext
1 let a = 4
2 let b = 7
3 let f x = x * x
4 let g x y = x + y
5
6 f (g a b) => 121
```

Let’s manually convert it to CPS style:

```plaintext
1 let a_mc cont = cont 4
2 let b_mc cont = cont 7
3 let f_mc cont x = cont (x * x)
4 let g_mc cont x y = cont (x + y)
5
6 // that can be called as:
7 b_mc (a_mc (g_mc (f_mc print_number))) // => 121
```

That was very easy, right? It follows from this simple example that there must be automated ways of doing this transformation (and they are actually as simple as what we see here). We won’t go into much details, because it would require us to define a language, etc, but you get the idea.

Why would a compiler do this? We’ve already seen the foremost benefit of automatic translating
your code to CPS-style: not having to worry about stack overflows. Considering that most functional languages depend heavily on recursion, it becomes a language feature to not have a limit on it.

So after transforming your code into continuation-passing style, the compiler is able to tail call optimize every single function in your program. Just remember that even in those languages there can still be positive performance side effects of making your recursive function tail-call. For example, rewriting things with accumulators will reduce the runtime of some algorithms even after tail call.

**Imma let you finish, but…**

Ok. Now that we are continuations rock stars, we can tackle a real problem.

A class of very well known problems (that you will for sure see again in the future) are the so called backtracking algorithms. Those are algorithms where you have to search a space of solutions by making guesses and being able to backtrack on those guesses once you reach a dead end. Ideally, you are able to backtrack way before you made all decisions you have to make (therefore pruning the search space).

The most well known backtracking problem is the n-queens problems. It goes like this: imagine a n by n chess board. Find a set up where you put n queens on the board that are not threatening each other (i.e., does not share the same row, column or diagonal). This is a solution for n=8:

```
One could think of an algorithm where you generate all possibilities of a board with n queens and then simply check if the condition is satisfied for each one of those arrangements. Unfortunately, this would take a really long time (there are 4,426,165,368 possible arrangements and only 92 solutions for n=8). That’s why backtracking is considered a good solution for n-queens: it allows us to skip most nonsense arrangements very fast. There are other optimizations one could do to solve this problem, but let’s keep it simple.

Let’s represent our board by a series of integers for each position, as if our board was linear array from 0 to n*n. For example, for a n=8 board:

```
 0  1  2  3  4  5  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63
```
Then, we are going to define a function that tells us if two queen positions are threatening each other:

```haskell
1  // Is @p1 a threat to @p2?
2  let is_threat n p1 p2 =
3      let (x1, y1) = (p1 % n, p1 / n)
4      let (x2, y2) = (p2 % n, p2 / n)
5      x1 = x2 || y1 = y2 || x1 + y1 = x2 + y2 || x1 - y1 = x2 - y2
```

Next, let’s define a function that tells us, given a position, would it threaten other queens we have already selected:

```haskell
1  // Is @p in conflict with any of positions on @queens?
2  let rec is_conflict n p queens =
3      match queens with
4      | [] -> false
5      | x::xs -> if is_threat n x p then true else is_conflict n p xs
```

Ok. Now we have all domain-specific (n-queens-specific) functions we need. Now let’s do the function that will search with backtrack until it gets all solutions using CPS.

This function will get a list of position to test, a list of already selected queens and a backtrack continuation. It will try each position, making sure our constrain (is_conflict) is satisfied. When it selects enough queens (n), it will record that solution and keep looking for others. When it decides that it wants to try a queen in a particular position, it will change the backtrack continuation to say: in case you can’t find a solution with this, use this continuation to keep going without the decision I just made.

```haskell
1  let rec search n clist moves back =
2      match clist with
3      | [] -> back ()
4      | (p::ps) ->
5          let new_backtrack = fun () -> search n ps moves back
6          if is_conflict n p moves
7              then new_backtrack ()
8          else
9              let new_moves = moves @ [p]
10             if List.length new_moves = n
11                then new_moves
12                else search n ps new_moves new_backtrack
```

That’s pretty much it. So let’s take a closer look at the search function. It’s trying all positions on clist (which is an array of positions). When there’s no more positions, it backtracks (line 3). Otherwise it gets the first position and sees if it can add it to the current queen position
(moves) without causing conflict (line 4-6). If it can’t then it skips this positions and keeps looking with the other positions (line 7).

If it can add that position, it does so (new_moves, line 9). If it has gathered enough queens (line 10) then it is going to return this particular selection of positions (line 11). Otherwise it has to keep looking for more queens, but it changes the backtrack continuation to backtrack the choice it just made (i.e., the addition of p on moves) (line 12).

That’s it. Then we can have:

```
let queens n = search n [0..n*n-1] [] (fun () -> [])
```

I.e., search all moves, starting with no queens. The default backtrack continuation does nothing. Calling this returns us all solutions for the n queens problem. For example, the one above was generated with:

```
queens 8
// ==> [0; 12; 23; 29; 34; 46; 49; 59]
```

With a little modification, we could make it return all solutions (instead of the first). Then we could do something like:

```
List.map (fun n -> List.length (queens n)) [1..10]
// ==> [1; 0; 0; 2; 10; 4; 40; 92; 352; 724]
```

which would shows us how many solutions there are for each n.