Ela

DYNAMIC FUNCTIONAL LANGUAGE

by Vasily Voronkov

WORK IN PROGRESS

April 2011

Version 0.1

Contents

Chapter 1

Introduction

1.1. What is Ela?

This book provides an overview of Ela programming language. Ela is a free (both noncommercial and open source) programming language that was implemented in C# and can run under CLR and Mono.

Ela is a dynamically typed strict functional language. It is an impure functional language unlike, for example, Haskell and does support side effects (but does not favor them) and even provides some imperative programming features.

Dynamic typing means that Ela doesn't type check in compile time but defers all type verifications until the run-time. In practice it allows compiler to behave in an optimistic manner – if an operation is syntactically and semantically correct than compiler thinks that the code is also correct and will let you to execute it. Such an optimism of Ela compiler gives you a lot of power – you are not burden anymore with the limitations of a particular type system. Also the code keeps short because you don't have to prove to Ela compiler that your program is correct and operate with types in an allowed manner. Ela compiler is gentle enough to believe you without it. Of course this approach has its own downfalls but comparison of static and dynamic typing is not the topic that we are discussing now.

It is important to understand however that Ela is pretty different from some popular dynamic languages such as JavaScript. Ela is not JavaScript. Ela does support dynamic typing but unlike JavaScript Ela

- doesn't use dynamic name lookup (if you try to reference a variable that is not declared an error is generated before the code gets executed)
- doesn't support dynamic scope but uses lexical scope (like such languages as C, C#, OCaml, Haskell, etc.)
- is not a weakly typed language (try to sum string and integer and you will get a compile time error)
- doesn't support *eval* in JavaScript style (that can capture local variables and can be used to create pretty weird side effects)

With Ela you will have much more compile time errors and static checking. Dynamic typing is a tool that is here to give you power, not problems.

Now for the *strict* part. Ela is a language with strict, or eager, evaluation.

It means that if you write code like so:

```
let x = 2 + 2
```

the right part of the declaration is executed immediately and x is initialized with the result of its evaluation.

This is not the case of non-strict languages like Haskell. In Haskell the same code will be executed pretty differently. An x will be initialized with the *expression* `2 + 2` and this expression will be executed only when its value is really needed, e.g. when you want to perform some operations with x.

Ela provides a support for lazy evaluation but in order to use it you have to explicitly tell the compiler that evaluation of a particular expression should be deferred until its value is needed. This is done like so

```
let x = (& 2 + 2)

let y = x * 2 //The '2 + 2' expression is evaluated here
```

In the contrary in Haskell you will have to explicitely tell the compiler that you want a certain expression to be executed in a strict manner. Haskell can use lazy evaluation by default because Haskell is a pure functional language with no side effects while Ela as we've already learned is not.

And now we come to the last part - functional. When I say that Ela is a functional language I mean that Ela is a declarative language in which all operations can be seen in the light of combinations of functions. It is probably easier to explain it by an example.

1.2. Imperative and functional

In an imperative language you have to code a sequence of actions instructing a compiler what it should do to finally come up with what you need. Let's take some simple task to illustrate it. Imagine that you need to find all numbers that are greater than 5. This is how your code might look like:

```
for (int i = 0; i < arr.Length; i++)
    if (arr[i] > 5)
        newArr.Add(i);
```

But what if next time you need all numbers that are greater than `10`? Or lesser than `4`?

In Ela you will first create a function that can filter a given list using the specified predicate. This is how this function might look like:

```
let filter _ [] = []
    filter f x::xs | f x  = x :: filter f xs
                   | else = filter f xs
```

A `filter` function takes a predicate and a list and constructs a new list with the elements from the initial list that satisfy the predicate.

Once we have a `filter` function we can define a more specific function that does the task:

```
let filter' = filter (\x -> x > 5)
```

Or even shorter like so:

```
let filter' = filter (>5)
```

That is all. We have a solution that can be easily reused in the cases when we need a different selection criteria, not just greater than 5.

OK, that is the place where I should try to convince you how readable and declarative the Ela solution is, while you (if you haven't tried the functional languages before) will probably think that this code looks like a cryptic mess in comparison with the imperative `for` cycle above.

But don't give up yet. I am sure that you are bold enough to not quit at the very beggining. Anyway it should be pretty interesting to understand why a lot of people think that this weird functional code is so much better than the imperative one. But you definetly have a lot of questions - what does this strange `filter` declaration with pipes and multiple equatations mean, why do we call `filter` with a single argument when it accepts two arguments, what is this is `\x -> x > 5` thing and, more importantly, what `(>5)` is doing here?

Lets start in order.

1.3. Recursion

The thing that worse to be mentioned first is recursion. You have definitely heard this term before as soon as there are many applications for recursion even in imperative languages. I am also confident that there is no surprise for you that all code with cycles can be rewritten without cycles using recursion. Let's write a recursive `filter` function in C#:

```
void filter(int i, int[] arr, List<int> newArr) {
   if (arr.Length == i)
     return;

   if (arr[i] > 5)
     newArr.Add(arr[i]);

   filter(i + 1, arr, newArr);
}
```

Now, this code has obvious problems in comparison with `for` cycle. Instead of simply looping through an array it calls a function to process every element which generates a serious pressure on a stack. Moreover this function will cause stack overflow if you try to process a huge array using it.

Taking all that it appears that cycle is a much better choice here. But there is one tiny problem – Ela doesn't have cycles. At all. As a result recursion is the only way to go in Ela. Does it mean that you always have to sacrifice performance when writing code in such a way? Not necessarily.

If you look closer at a `filter` function definition you can notice that a function call is the very last instruction in the function body. This is called *tail recursion*. In fact in such cases it is pretty trivial to optimize this functional call away and to translate the code into a simple cycle. And that is what Ela compiler does. You can use recursion which is more declarative (at least in Ela) with no performance hit. And no chances of stack overflow.

1.4. Anonymous functions

An expression `\x -> x > 5` is used to declare an anonymous function which is useful when you don't really need to bind a function to a particular name but to provide it as an argument for another function like in our example. Of course you can declare a function using this syntax *and* bind it to a name like so:

```
let fun = \x -> x > 5
```

But this is exactly the same as:

```
let fun x = x > 5
```

Also as soon as all functions are first class values (like integer numbers or strings) you can also use a named function as an argument of another function:

```
let fun x = x > 5

let filter' = filter fun
```

But in such cases it is probably more visual to declare such one-time-to-use functions in place.

1.5. Closures

Another important thing to understand about Ela functions is that all functions in Ela can capture names declared in the parent scope, e.g. this is a perfectly valid code in Ela:

```
let x = 2

let fun y = x + 2
```

The function `fun` captures the name `x` – not just the value of the `x` – which can be seen in the following example:

```
let rec = {!x=2} //Here we declare a mutable record

let fun y = rec.x + y

rec.x <- 4 //Change the value of x

let res = fun 2 //The result is 6
```

1.6. Partial application

And the final nuance – all functions in Ela actually accept one and only one argument. I really mean it. There is no way to declare a function that takes, say, two arguments or zero arguments. You might wonder how we've managed to write a `filter` function above which seem to have two arguments instead of one. But don't hurry up.

When we decide that all functions in our language have a single argument it changes quite a few things in the language itself. First we no longer need braces (like in C#) to enclose the argument list in a function call – as soon as we don't have any argument list at all, just a single argument.

You might ask how do we know what is an argument of our function when this argument is not a simple literal value or a variable.

In C# the code `sin(x) * 2` means that we need to apply `sin` function to the value of `x` and then multiply the result of this application by two. The equivalent code in Ela `sin x * 2` looks like we need to apply `sin` to the value of `x * 2`. But in reality this code is executed exactly in the same way as C# code before. This is because function application binds tigher than most of other operators. And if you want to apply `sin` to the result of multiplication of `x` by `2` you can enclose this expression in braces, e.g. `sin (x * 2)`.

Also function application is left associative which means that if you see an expression like `sum 2 3` this a perfectly valid code and it is executed in same way as `(sum 2) 3`. In other words here we call a function `sum` with an argument `2` and the result of this application is another function that is called with an argument `3`. So `sum 2 3` is not a single function call with two arguments but two function calls.

And this behavior actually unveils the way how Ela deals with the lack of functions that accept multiple arguments. We simply declare a function that returns another function (which in its turns might also return another function and so on).

This is how an implementation of a `sum` function might look like in Ela:

```
let sum = \x -> \y -> x + y

let res = sum 2 2 //Obviously 4
```

or

```
let sum x = let sum' y = x + y
            in sum'
```

or

```
let sum x = sum'
            where sum' y = x + y
```

This trick is possible because as it was mentioned above all functions in Ela are closures and capture variables declared in the parent scope. The same code in C# looks like so:

```
Func<int,int> sum(int x) {
    return y => x + y;
}
```

See how the nested function captures an `x` parameter from the scope of the parent `sum` function? A pretty similar staff happens when you declare an Ela function like `\x -> \y -> x + y`.

But frankly speaking all these declarations look quite cumbersome. Its hard to argue that this single-argument-function concept finally leads us to a pretty wordy and inconvinient syntax. But we haven't failed yet. The syntax is not good for many cases? So lets sweeten it with some syntax sugar:

```
let sum x y = x + y
```

I believe it looks much better now. And it works for anonymous functions as well. We simply say that a declaration in a form `\x y -> x + y` is a syntax sugar for `\x -> \y -> x + y`. We still have two functions here not just one but sometimes it is easier to think of such functions as of a single function like you do in languages like C#.

Hint

For convenience I will stick with the regular terminology. For example I will be referring to a function like `sum` as of having *two* arguments. But we of course know what really happens under the hood.

But the main question is still unanswered. What was the reason to invent this whole one-argument-function thing? Is it just because somebody really hate braces? Of course there are reasons much better than that.

The main idea behind representation of a function with multiple arguments as a chain of nested functions with single argument is an ability to partially apply functions. Let's say that you implement a `sum` function in C# language:

```
int sum(int x, int y) {
  return x + y;
}
```

It looks good. But what if we need another function for just one argument that always add this argument to some fixed number such as 5?

You basically have two options here: to implement a new function that mostly duplicates the previous `sum` definition or to manually hard code this fixed argument in every call to the `sum` function. Both options don't really feel right for me. In Ela, having the `sum` function defined in a similar way, you can create a new function by providing only the first argument to it:

```
let sum x y = x + y

let sum5 = sum 5
```

This technique really gives you a lot of power – it allows you to define generic functions and then create specialized versions of them "on the fly" by fixing some of their arguments.

1.7. Operators as functions

Let's return to the `filter` function. As you can see the original definition of `filter` accepts two arguments – a predicate function which is used to choose which elements from the source list should be included in the result and the list itself. In other words by changing the predicate function we can filter any list using any condition we can ever think of. So the `filter` function is really generic. However if you want a function to filter a list with a particular condition (e.g. to obtain all elements that are greater than five) you can easily create this function by providing the first argument without the second:

```
let filter' = filter (\x -> x > 5)

//or
//let filter' = filter (>5)
```

But what is this second notation where I use some strange expression `(>5)` instead of anonymous function declaration `\x -> x > 5`?

The trick here is that most of standard Ela operators are functions. The only difference between an operator such as `(>)` and a function is that operator is called by default using *infix* notation. Infix means that the operator reference is placed right in the middle of its arguments (when a regular *prefix* notation assumes that we place a function before its arguments). Actually all Ela functions can be called using infix notation – you just need to enclose them in back apostrophes:

```
let res1 = sum 2 2 //prefix call

let res2 = 2 `sum` 2 //infix call
```

There is no real difference between the two and of course you don't have to use the infix form with regular functions if you don't like it – however in certain cases infix form might be a little bit more readable.

Operators in their turn can also be called in prefix form just as all other functions – you just need to enclose them in braces:

```
let res3 = 2 + 2

let res4 = (+) 2 2
```

As soon as operators are just functions with a different default call notation the technique of partial application described above is possible with operators as well. A code like `(+2)` fixes the second argument of the `(+)` operator and the code like `(2+)` fixes the first argument. The same for comparison operators and many other operators.

So `(>5)` is just a neat and more readable shortcut for the explicit function declaration `\x -> x > 5`.

1.8. Pattern matching

Now we undestand how functions work, that they are first class values like other data types, that they can be partially applied, that operators are also functions – however the declaration of `filter` function might still look pretty cryptic. And now the mystery is unveiled – the `filter` function is defined using *pattern matching*.

Pattern matching is a powerful technique adopted by many functional languages. You can think of pattern matching as of more concise and declarative replacement for the sequence of `if` statements to which you got used to in imperative languages. The idea behind pattern matching is as simple as all genius things.

Normally objects in programming languages are constructed using some special syntax or operators. For example, integer numbers are constructed using integer literals (e.g. `42`), strings are constructed using string literals (e.g. `"foo"`), tuples are constructed using tuple literal (that is a sequence of elements separated by commas and enclosed in parenthesis) and linked lists are constructed using list construction operator:

```
let lst = 1::2::[] //[] is a an empty list
```

Hint

List is basically a "functional" replacement for an array that can be used instead of it in many cases. Lists are just linked and immutable sequences of elements. In the example above we have an integer number 1 which is wrapped in a special structure that holds the value 1 and a reference to the next element, which is again wrapped in a special structure that holds number 2 and also points to the next element with is an empty list. The first element of a list is usually called a *head* and the rest of the list without the first element (the list with its head chopped off) is called a *tail*. That is all that you need to know for now, there is a separate chapter about lists in this book.

The genius idea that I mentioned above is to use object construction syntax to not only construct objects but also to deconstruct them, to disassemble them in parts. Let's see how it works through code sample:

```
//Creating a tuple with three elements
let tup = (1,2,3)

//The value of res is 6
let res = match tup with
                (x,y,z) = x + y + z
```

A mere equivalent code in C# will be the following:

```
var arr = new int[] { 1,2,3 };
var res = 0;

if (arr.Length == 3)
    res = arr[0] + arr[1] + arr[2];
```

I hope you will agree with me that Ela version is much more concise than the C# code. Moreover C# code is pretty error prone – it requires us to declare `res` as a mutable variable (because `if` is a statement and not an expression) even if we don't need to change its value afterwards. We have to manually test for the array length and explicitly specify indices of its elements. If you make a typo or just peek an incorrect index than the compiler won't be able help you at all and you will end up with a run-time error.

Pattern matching is such a neat thing that it is used almost everywhere in Ela. If you see some construct and you don't know what is it – it is most likely yet another way to do pattern matching. For example, the code above can be written differently like so:

```
let res = x + y + z
          where (x,y,z) = tup
```

Here we use pattern matching directly in variable declaration construct and it is even more visual than the example above. (If you don't understand some syntax in examples – don't worry, we will deal with these things later).

Linked lists can be pattern matched in the same manner as tuples – using list construction syntax:

```
let lst = 1::2::3::[]
//or you can use an equivalent list literal as a shortcut:
//let lst = [1,2,3]

let res = match lst with
```

```
                x::y::z::[] = x + y + z

let res2 = x + y + z
           where (x::y::z::[]) = lst
```

But what happens if you try to match an empty list or a list with just two elements using pattern `x::y::z::[]`? The answer is pretty obvious – a *match failed* run-time error. In order to avoid it you should take into account all the possible situations:

```
let res = match lst with
                x::y::z::[] = x + y + z //or [x,y,z] = x + y + z
                x::y::[]    = x + y     //or [x,y] = x + y
                x::[]       = x         //or [x] = x
                []          = 0
```

Here we have several match entries separated by semicolons. The first matches against a list with 3 elements, the second – against a list with two elements, the third needs only one element and the last matches an empty list. But in reality we still don't take into account all the possibilities. What will happen if `lst` contains four elements? A run-time error. Let's refactor our code:

```
let sum lst = match lst with
                    x::xs = x + sum xs
                    []    = 0

let res = sum lst
```

I hope this one looks better than the example above. And it also works much better. We have created a function that sums all elements in the list of any length. The pattern `x::xs` matches a list that contains at least one element (in this case `x` is bound to this element and `xs` – to the empty list `[]` which is usually called a *nil* list).

Our new function is a recursive function – it chops the head of a list and calls itself with the rest of the list. It is executed like so:

```
1 + sum [2,3]
2 + sum [3]
3 + 0
```

The result is obviously `6`. But let's compilate our task – now we need a `sum` function that sums only positive numbers and ignores the negatives. This is how we should change our code to fullfil this new task:

```
let sum lst = match lst with
                    x::xs | x >= 0 = x + sum xs
                    x::xs | x < 0  = sum xs
                    []             = 0
```

Now we are really close. This "piped" appendix to the pattern is called a *boolean guard*. It is nothing more than an old `if` statement in a new form. Here we simply specify that the first match entry should be selected when the head of the list is equal or greater than `0`, otherwise we need to skip to the second match entry where we ignore the head of the list and continue to unfold the rest of the list.

There are however a couple of things that just don't feel right about this function. First in the second match entry we bind the head of the list to the x when we don't really need it. In other words we declare a variable that is never used. It is not a good practice even in imperative languages. Let's fix it:

```
let sum lst = match lst with
                    x::xs | x >= 0 = x + sum xs
                    _::xs | x < 0  = sum xs
                    []             = 0
```

The pattern _ is a special "throw away" pattern. He we say that yep, we know that there is something here, but we don't need it so get rid of this staff please.

However our pattern matching construct still looks a bit wordy. Even with the "throw away" pattern we still repeat almost the same pattern twice. In order two avoid this kind of repetition Ela allows you simply to omit the pattern declaration in such cases:

```
let sum lst = match lst with
                    x::xs | x >= 0 = x + sum xs
                          | else   = sum xs
                    []             = 0
```

Now `sum` function looks OK but is still not perfect. We have to repeat the `lst` parameter twice. Also the whole `match` declaration syntax adds some unneccessary "weight" to the function. That is the reason why Ela supports yet another syntax sugar for pattern matching. Remember the pattern matching inside a variable declaration? This is a pretty similar case. You can use pattern match directly in function definition:

```
let sum [] = 0
    sum x::xs | x >= 0 = x + sum xs
              | else   = sum xs
```

Now it is perfect. You can see that here we use patterns instead of function arguments. It allows us not to declare any intermediate names (like `lst` in the previous example) that are required if you use a regular match construct.

But let's get back to the `filter` function in the initial example and take a look at it again:

```
let filter _ [] = []
    filter f x::xs | f x  = x :: filter f xs
                   | else = filter f xs

let filter' = filter (>5)
```

Does it still look cryptic?

1.9. Organization

I hope now you can make your mind whether it is worth to give Ela a try.

We have reviewed only a single simple function – and already discovered so many interesting features. Imagine what you would see next. Don't worry if you don't understand some of the concepts briefly mentioned above – we will return to them multiple times in the next chapters.

This book is organized as a guide to Ela and functional programming. It might be beneficial for you to read it even if you are not planning to use Ela right away – Ela doesn't invent a wheel and most of the things discussed here are typical for the majority of the functional languages.

This book is not a reference manual but can be used in such a way also. In the end of the book you can find a number of appendices that describe a language in a more formal manner.

I assume that a reader doesn't necessarily know other functional programming languages but has certain experience with imperative languages such as C#. In some cases I will point out the differences between Ela and C# and use code samples in C# to explain some of the Ela features.

1.10. Convention

I will use the following typographical convention in this book:

All important notions such as concepts in computer science are printed in *italics*.

All code samples including separate code blocks and in-line code (expressions, literals and variable names) are printed using `monospace` font.

When I refer to operators (like in a statement: operator `(::)` is right associative) operators are always printed in monospace font and enclosed in parentheses. Parentheses are not the part of operator syntax.

Chapter 2

Installation and usage

2.1. Obtaining Ela

You can download the latest Ela binaries from Google Code repository. Source code is available as well[1].

Ela distribution includes Ela implementation (which is presented as a single `ela.dll`), Ela Console tool (`elac.exe` that can be used to evaluate Ela expressions, compile Ela code into object files and to work in interactive mode) and standard library.

2.2. Prerequisites

In other to run Ela you need either Microsoft .NET Framework version 2.0 or higher or Mono version 2.6 or higher. If you choose to stick with Mono than you have to use Mono application launcher (`mono.exe`) in order to start Ela command line tool like so:

```
mono.exe c:\ela\elac.exe
```

Or in a Unix-like system:

```
mono \ela\elac.exe
```

In these examples I am assuming that Ela is installed in the directory `c:\ela` (Windows) or in a root directory `\ela` (Unix).

On Unix systems Mono is the only possibility to run Ela. On Windows system you can use both Mono and .NET Framework.

Ela works fine with later versions of .NET and Mono. For example, if you only have .NET 3.5 or .NET 4.0 you don't have to downgrade to a previous version.

Ela is compiled using *AnyCPU* configuration – it means that if you have a 32-bit operative system Ela will run as a 32-bit process but on 64-bit operative systems it will run as 64-bit process.

The minimum supported version of Windows is Windows 2000 SP3 (for the supported Unix versions please refer to Mono documentation). If you are running Windows Vista or higher you already have all the prerequisites you need – in Vista and later versions of Windows a required

[1] Download list: http://code.google.com/p/elalang/downloads/list

version of .NET Framework is already included by default.

2.3. Installing Ela

Ela itself doesn't require installation and/or configuration at all. As it was mentioned before the whole language implementation is a single `ela.dll` library that can be installed simply by copying files. `Ela.dll` is a fully managed .NET assembly and basically everything you need if you want to redistribute Ela with your applications.

The command line utility Ela Console doesn't require installation as well however you might need to spend a minute or two to do some optional configuration which can simplify the usage of interpreter.

Ela Console uses a standard application configuration file for settings which is called `elac.exe.config`. It already comes with a default one which might look like so:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="ref" value="lib\" />
    <add key="stdLib" value="elalib" />
    <add key="prelude" value="Prelude" />
  </appSettings>
</configuration>
```

The path `lib\` is a directory (relative to the location of core Ela files) where you have standard library modules. By default this path uses Windows convention; on Unix you might need to update it to use Unix path format. The `ref` key is used by Ela linker to determine which directories can be used to lookup referenced modules. You can specify several directories separated by semicolon like so:

```
<add key="ref" value="lib\;samples\" />
```

Ela standard library is partially implemented in C# (and partially in Ela itself). The `stdLib` key is used to tell Ela linker how the assembly with the "C# part" is named. It saves you some key strokes and instead of writing a module open directive like so

```
open Array#elalib
```

with the module name and DLL name after the hash sign you can write it like so

```
open Array
```

which is a little bit shorter.

Both of these keys are not required and Ela will be able to work without them (even if you fully delete configuration file) however it will look for the referenced modules only in the directory where your executable file is located and in the directory where Ela itself is installed.

The last key `prelude` is used to specify the name of a `Prelude` module. This module contains definitions of all basic Ela operators (such as arithmetic operators, bitwise operators, etc.) and standard functions. Without it you will have to define all these operators by yourself otherwise even code like `2 + 2` will fail to compile.

`Prelude` module is imported automatically if the `prelude` key is set to the name of a module. You always have a possibility to remove this setting from configuration file (to run Ela without `Prelude`) or to provide your own implementation of `Prelude` module that might use different symbols for operators, with different priorities, etc.

You can also specify other configuration settings via this file. In order to learn how Ela Console can be configured simply lunch it with the `-help` key. You will see a list of command line options – all of these options can be provided via configuration file as well. For example, lunching Ela Console like so

```
elac.exe -ml
```

is the same as having the following key in the configuration file:

```
<add key="ml" value="true"/>
```

And that is all we need. Ela is installed.

2.4. Ela files

Ela uses two types of files.

The first type has an extension `.ela` (e.g. `Core.ela`, `Prelude.ela`). Such files contain Ela source code. It is highly recommended to use UTF-8 encoding for them.

The second type has an extension `.elaobj` (e.g. `Core.elaobj`). These are so called *object files* that contain a binary representation of a compiled Ela program. They are not indented to be edited manually but you should be aware of their existence.

A typical case is to have a source code file (such as `Core.ela`) and an object file (`Core.elaobj`) right next to it. Ela linker will see that there is already a precompiled version of your module and won't rebuild it one more time (unless the date tag of source code file is newer than the date of an object file).

2.5. Using interactive console

Ela Console is a relatively easily tool to use.

If you don't specify a file name to execute, Ela Console is lunched in an interactive mode. You should see the following banner (the text may vary depending on the Ela version and run-time environment):

```
Ela Interpreter version 0.9.0.0
Running CLR 2.0.50727.4952 32-bit (Microsoft Windows NT 6.1.7600.0)

Interactive mode
Enter expressions and press <Return> to execute.

ela>
```

Interactive mode is a really neat thing which is very handy when you want to test some language constructs or debug your programs. Just type Ela expressions and press `Return` key to execute them:

```
ela>"Hello, world!"

Hello, world!

ela>12 + 2 * 4

20

ela>(1,3) + (12,3)

(13,6)
```

But that is not the only cool thing.

With interactive mode you can compose your programs chunk by chunk. Ela interpreter "remembers" all the previous declarations you did:

```
ela>let x = 2

ela>let y = x + 2

ela>y

4
```

Hint

As you can see you can simply check which value is bound to a particular name by typing this name and pressing `Return`.

While in interactive mode if you enter code that doesn't even compile you will see an error message (or even several messages and warnings if you did something really bad) and of course such code is never executed:

```
ela>let z = x1 + 2

(1,9): Error ELA611: A name 'x1' is unresolved. It is not declared
externally or its type has changed.
```

As a result the name `z` wasn't declared which you can double check by entering `z` in console:

```
ela>z

(1,1): Error ELA611: A name 'z' is unresolved. It is not declared
externally or its type has changed.
```

However if you write code that fails at run-time like so

```
ela>let z = 12 / 0

(1,14): Error ELA803: Division by zero of value '12' of type 'int'.
        in <memory> at line: 1, col: 14
```

a name z actually gets declared – this is just initialization block that fails with an error. So we have a declared name which is not initialized – and that is not an allowed situation in Ela. In order to deal with this problem Ela Interactive initializes z with `unit` (if you don't understand what is unit, you can think of it as an Ela replacement for `void`).

Hint
Small hint - enter `#clear` and hit `Return` if you want to clear the messages in console.

Interactive mode doesn’t have any restrictions – all the expressions and statements that you can write in regular Ela files are valid in interactive mode as well. For example you can declare functions:

```
ela>let sum x y = x + y

ela>sum

sum:*->*
```

Ela uses an indentation based syntax – that is when code blocks are specified using indents (instead of curly braces like in C#). Writing code directly in console doesn’t create much trouble if you only need to declare trivial functions. However even if you want to evaluate a more complex expression it still has to be packed in one line. Basically there are two options here.

First you can use semicolon character to separate different code blocks. This is how our filter function can look if we define it in such a manner:

```
ela>let filter f x::xs | f x = x :: filter f xs | else = filter f xs;
filter _ [] = []
```

But you can probably agree with me that it is not the best to declare functions. And that is why there a second option available – you can lunch Ela Console with `-ml` key:

```
Ela Interpreter version 0.9.0.0
Running CLR 2.0.50727.4952 32-bit (Microsoft Windows NT 6.1.7600.0)

Interactive mode
Enter expressions in several lines. Put a double semicolon ;; after
an expression to execute it.


ela>
```

Now you can define functions using indentation based syntax and the code gets evaluated only if it ends with a double semicolon:

```
ela>let filter f x::xs | f x = x :: filter f xs
--->                   | else = filter f xs
--->    filter _ [] = [];;
```

You can even switch between different editing modes without restarting Ela Console – just type #ml to toggle multiline mode:

```
ela>#ml

Multiline mode is off
```

Hint

In a case if you forget some commands you can always display a help file even if you are in interactive mode – just type `#help` and hit `Return`. You don't need double semicolons for all commands that start with # – even if you are in multiline mode.

We will use both single and multiple line modes. The first is good for simple expression and doesn't require you to terminate each entry with a double semicolon. The second is better for complex constructs such as function declarations. I will tell in which mode you should lunch Ela Console at the begging of each chapter.

2.6. Compiling Ela code

Ela command line tool also allows you to compile Ela source code files into object files. This is done by appending a `-compile` switch after the file name like so:

```
c:\ela>elac samples\myfirstmodule.ela -compile
Ela Interpreter version 0.9.0.0
Running CLR 2.0.50727.4952 32-bit (Microsoft Windows NT 6.1.7600.0)

Compilation completed. File 'c:\ela\samples\myfirstmodule.elaobj' created.
```

By default Ela compiler will create an object file of the same name as source code file (with an extension changed to `.elaobj`) and place in the same directory. However you can override the default behavior and specify the output file explicitly like so:

```
c:\ela>elac samples\myfirstmodule.ela -compile -out c:\mymod.elaobj
Ela Interpreter version 0.9.0.0
Running CLR 2.0.50727.4952 32-bit (Microsoft Windows NT 6.1.7600.0)

Compilation completed. File 'c:\mymod.elaobj' created.
```

Sometimes it might be useful just to browse the Ela byte code without saving it to file. Ela command line tool provides this possibility as well with the help of `-eil` switch. You can print out to console the byte code generated for a particular module or even lunch an interactive mode with `-eil` switch:

```
c:\Projects\Ela>elac -eil
Ela Interpreter version 0.9.0.0
Running CLR 2.0.50727.4952 64-bit (Microsoft Windows NT 6.1.7600.0)

Interactive mode
Enter expressions and press <Return> to execute.

ela>2 + 2

EIL (0-3):
00: PushI4 2
01: PushI4 2
02: Add
03: Stop

ela>4 / 2

EIL (4-7):
04: PushI4 4
```

```
05: PushI4 2
06: Div
07: Stop

ela>
```

This is useful when you want to understand how a certain block of code is executed and to dig deeper into the language. But of course you don't have to know Ela assembly (called *EIL* for *Ela Intermediate Language*) to successfully program in Ela.

Chapter 3

Simple expressions and syntax

3.1. Simple expressions

Now that you have learned how to use Ela interactive console we should definitely see it in action.

Lunch `elac.exe` without parameters and try to enter some expressions into it:

```
elac>2 + 3 * 4

14
```

OK, it works. As you can see default operator priority in Ela is pretty similar to that of C style languages such as C#. However unlike C# Ela is an expression centric language. What does it mean?

It means that almost everything in a language is an expression and yields a value. Moreover a whole program in Ela can be seen as a single expression.

> **Hint**
> Of course not all of the language constructs are expressions as soon as some of them cannot return any meaningful value and will even look pretty weird if you try to use them in the same context as you use expressions. A good example is an `open module` directive.

Ela interactive console doesn't provide any special mode that allows you to directly evaluate expressions – it simply follows the rules of the language according to which `2 + 3 * 4` is a completely valid Ela program. You can copy and paste this to a file and execute it. Even `5` or `"Hello, world!"` are valid Ela programs that evaluate to themselves.

Ela syntax has very much in common with syntax of C style languages. Ela uses similar literals for strings and chars (including escape codes). Ela supports all basic arithmetic operators and use parenthesis as well as C# for grouping expressions. This is how we can change the above code sample to produce a different result:

```
elac>(2 + 3) * 4

20
```

When you write such expressions, Ela code is not much different from C# or JavaScript. But it does have a very important difference from the latter. Try the following in console:

```
elac>12 + "42"
```

```
(1,6): Error ELA826: A value '42' of type 'string' doesn't support
operation 'add'.
        in memory at line: 1, col: 6
```

What have just happened? Well, that's a separate story.

3.2. Strict and dynamic

Ela is a strictly typed language: it only allows you to deal with the types in a way that is explicitly supported by these types. For example, numbers support arithmetic but strings don't. As a result you cannot perform arithmetic operations on strings and numbers. In fact such an operation – addition of an integer to a string – is completely impossible and the only way to produce some kind of a result is to convert one of the operands to another type. You can make a number from string or a string from number. The last one is what JavaScript does. Ela however restricts implicit conversions in such a case and instead of weird `"1242"` you will get a run-time error.

But wait a second. Why a *run-time* error? Isn't it the case that should be normally tracked down by a compiler?

Yes, normally – by a statically typed language. But Ela is not the one of the kind.

Dynamic typing does render certain peculiarities of the language. For example a single expression in Ela may have different types depending on the run-time conditions. I can illustrate it using `if` operator:

```
if someValue then 12 else "42"
```

Imagine that `someValue` is a variable that is defined before and may have different values depending, for example, on the user input. As a result sometimes an expression above may have an integer type and sometimes it can be a string.

> **Hint**
> Yes, `if` operator in Ela is an expression unlike such languages as C#. As soon as it is an expression and should always yield a value an `else` clause is required and omitting it is a syntax error.

However this is usually not a good style of programming even in Ela – there are much better ways to do the same thing but we will deal with them later.

Let's move forward. So for now we have seen expressions of two types – integer numbers and strings. In this chapter I will describe all simple primitive Ela types including four numeric types, strings, chars, booleans and unit.

3.3. Primitive types

3.3.1. Booleans

Most of modern programming languages have a notion of a *boolean* data type (which is called `bool` in Ela) and Ela is not an exception here. Boolean type is pretty straightforward. Instances of this type can be either `true` or `false` – with no third option.

You can create a boolean value using the literal form. Try the following in Ela Console:

```
ela>true

True

ela>false

False
```

Ela doesn't allow implicit conversions between booleans and integers therefore the following code is not correct:

```
ela>if 0 then "zero" else "not zero"

<memory>(1,1): Error ELA826: A value '0' of type 'int' doesn't support
operation 'bool'.
        in memory at line: 1, col: 1
```

An equivalent to the *boolean not* (`!`) operator in C# is an ordinary `not` function which is defined in `Prelude`:

```
ela>not true

False

ela>not false

True
```

Boolean type support equality operations such as *equal* (`==`) and *not equal* (`<>`):

```
ela>true == false

False

ela>true <> false

True
```

As you can see from this example a result of equality operation is a value of a boolean type. Therefore we can change the erroneous code above like so:

```
ela>if 0 <> 1 then "zero" else "not zero"

zero
```

The result of comparison operations such as *greater* (`>`), *lesser* (`<`), *greater or equal* (`>=`) and *lesser or equal* (`<=`) is also a value of a boolean type, however boolean itself doesn't support comparisons. Which is perfectly logical as soon as there is not much sense to test whether `true` is greater than `false` or vice versa.

Ela has a set of operations that are predefined for boolean types. First of all these are *boolean and* (`&&`) and *boolean or* (`||`) operators. They work exactly the same way as in C, C#, Java, etc. It is important to understand that these operators are lazy (which is also a typical behavior in other languages as well). Operator `(&&)` returns `true` if both operands are `true` and operator `(||)` returns `true` if at least one of its operands is `true`:

```
ela>if true && false then "Yes" else "No"

No

ela>if true || false then "Yes" else "No"

Yes
```

Lazy here means that these operators are executed like so:

```
ela>if true then (if true then "Yes" else "No") else "No"

No

ela>if true then "Yes" else if true then "Yes" else "No"

Yes
```

As soon as `(&&)` yields `true` if both operands are `true` there is no need to evaluate both operands if we see that the first doesn't satisfy the condition. The same for `(||)`. If we see that the first operand satisfies the condition there is no need to evaluate the second operand as soon as it won't affect the result anyway. It may come up pretty handy because you can have complex expressions as operands (such as costy function class, etc.).

The last thing to mention about boolean operators is that unlike many other operators in Ela these two are not functions but just special forms. They are not defined in `Prelude` and cannot be overridden. If you try to use them as functions you will get a compile time error. This is because Ela is a language with strict evaluation and it is not possible to implement a function with two parameters and transparent behavior that will evaluate the second parameter only when needed.

3.3.2. Numeric types

Ela has four built-in numeric types – signed 32-bit integers (`int`), signed 64-bit integers (`long`), single precision floating point numbers (`single`) and double precision floating point numbers (`double`).

> **Hint**
> As a remark that you might find useful - all these types are directly mapped to .NET data types: `System.Int32`, `System.Int64`, `System.Single` and `System.Double`.

This is how you create values of these types:

```
ela>42

42

ela>42L

42
```

```
ela>1.42

1.42

ela>1.42D

1.42D
```

You have probably noticed the postfixes `L` and `D`. A postfix `L` is used to denote that you need a value of type `long` and `D` – that you need `double`, not just `single` which is used by default for floating point numbers. However you can also use postfix F for single precision floats – for example if you want a value without fractional part to be treated as fractional:

```
ela>42f / 5

8.4

ela>42 / 5

8
```

Numeric types are somewhat different from other data types. That is the place where Ela run-time environment is not too strict for the sake of convenience and does support some implicit conversions from one type to another. The rules for implicit conversions are pretty straightforward. In order to better understand them I will put all the numeric types in order like so:

- `int`
- `long`
- `single`
- `double`

And that is what happens – `int` can be implicitly converted to `long`, `single` and `double`, `long` – only to `single` and `double`, `single` – just to `double` and life is not fair to `double`, it cannot be converted to any other numeric types at all. That is because this "conversion chain" doesn't work backwards – `long` is never implicitly converted to `int`, `single` – to `long`, etc.

Here is a small example of how implicit conversions work:

```
ela>12 + 24.2

36.2

ela>25 / 12

2

ela>25D / 12

2.08333333333333

ela>4.3 * 120
```

```
516
```

As you can see numeric types support all basic arithmetic operations including *addition* (`+`), *subtraction* (`-`), *multiplication* (`*`), *division* (`\`), *remainder* (`%`), *power* (`**`) and *negation* which is written (`--`) unlike many other languages where negation uses the same identifier as binary subtraction.

These operators have the same priority as in C style languages. *Power* and *remainder* operators have the same priority as *multiplication* and *division*:

```
ela>14 % 5

4

ela>3 ** 3

27

ela>2 + 4 ** 3

66

ela>12 - 40 % 3

11

ela>155 * 20 / 2

1550
```

A couple of other samples to illustrate negation operation:

```
ela>-42 //This is a partial application of a '-' function

<f>:*->*

ela>--42 //Finally, a correct negation syntax

-42
```

Types `int` and `long` also support bitwise operations such as *bitwise and* (`&&&`), *bitwise or* (`|||`), *bitwise xor* (`^^^`), *left shift* (`<<<`), *right shift* (`>>>`) and *bitwise not* (`~~~`). These have the same behavior as equivalent operators in C# however their priority is different:

```
ela>let sn = 48 ||| (134 <<< 8)

ela>sn

34352

ela>sn >>> 8

134

ela>sn &&& 255

48
```

Numeric types also support a bunch of other operations. They can be tested for equality using operators (`==`) and (`<>`), compared using operators (`>`), (`<`), (`>=`) and (`<=`). They also support operations successor and predecessor (`succ` and `pred`) and taking a minimum and a maximum value (`min` and `max`):

```
ela>succ 42

43

ela>pred 42

41

ela>max 1

2147483647

ela>max 1L

9223372036854775807

ela>min 1L

-9223372036854775808

ela>max 1D

1.79769313486232E+308
```

The functions `succ`, `pred`, `max` and `min` are standard Ela functions defined in `Prelude`.

Ela doesn't perform an overflow check so you should be careful with your calculations. We are running a few steps forward but here is simple example of factorial calculation that overflows – just to illustrate the idea:

```
//naive implementation of factorial function
ela>let factorial x = if x == 0 then 1 else x * factorial (x - 1)

ela>factorial 13 //works OK

1932053504

ela>factorial 20 //overflows

-2102132736
```

3.3.3. Strings and chars

Strings and chars are yet another couple of data types that should be well known from other programming languages.

Strings in Ela are unicode sequences of characters. In the current implementation they do map directly to the .NET Framework data type `System.String`. This is how you can create a string in Ela:

```
ela>"Hello, world!"
```

```
Hello, world!
```

Strings in Ela (as well as chars) support C style escape codes. For example this is how you can create a two line string with double quotes:

```
ela>"Hello,\r\n\"Basil\""

Hello,
"Basil"
```

Also strings are immutable – you cannot change them in-place.

Unlike many programming languages, strings in Ela don't support `(+)` operator. This one is reserved in Ela only for arithmetic. You can use another polymorphic operator `(++)` which is called a *concatenation operator* if you want to concatenate strings:

```
ela>"Hello"++ "\r\n" ++ ", world!"

Hello,
world!
```

As soon as strings are immutable, concatenation operation always produces a new string. Also this operation doesn't perform implicit type casts – its both operands should be strings otherwise a run-time error is raised.

Ela supports both single line and multiline strings. The equivalent for the latter in C# are verbatim strings. However Ela uses a different literal for multiline strings:

```
<[multiline string
     with
         "quoted" word]>
```

Multiline strings don’t support escape codes (exactly as verbatim strings in C#).

Now, what you can do with strings. You can test strings for equality and compare them using comparison operators:

```
ela>"Hello" == "world!"

False

ela>"Hello" > "world!"

False

ela>"Hello" <= "world!"

True
```

You can also use built-in `length` function to calculate string length:

```
ela>length "Hello, world!"

13
```

You can use an indexing operator to obtain a char at a given offset like so:

```
ela>"Hello".[0]

H

ela>"world!".[5]

!
```

Indexing operator in Ela is pretty similar to that in C# with the only exception that it uses a extra dot before the square braces.

> **Hint**
>
> If you already get used to the indexer in the form `newList[2]` it is really important to remember that this dot before the square braces is required. This is because `newList[2]` is also a valid syntax in Ela but has a completely different meaning. As you remember Ela uses a simple juxtaposition for a function call, e.g. `x y` in Ela means that we apply an function `x` to the argument `y`. The same thing happens with `newList[2]` expression which is understood by Ela as: apply a function `newList` to the expression `[2]` (which in its turn evaluates to a list with a single element).

As it was mentioned above when applied to a string indexing operator returns a value of type `char`. Char is a pretty basic data type that is used to represent a single unicode character. It also has its own literal in Ela similar to the one that is used in C style languages:

```
ela>'H'

H
```

Chars as well as strings can be tested for equality and compared. Also chars support built-in `succ` and `pred` functions:

```
ela>succ 'a'

b

ela>pred 'b'

a
```

These are basically all the things that you can do with chars. Unlike some other languages chars in Ela are not numeric types and cannot be implicitly converted to numeric types. Therefore code like `'a' + 'b'` is errorneous in Ela and won't work. You will have to explicitly convert chars to integers in order to fix it.

3.3.4. Unit

Unit is somewhat similar to the notion of `void` from C#. It is different in certain aspects however.

In Ela `unit` is a real data type and you can create a value of this type. This is a first class value in a sense that you can pass it as an argument to a function, return it from a function, add it as an element to a list, etc. Unit even has it own literal form in Ela. This is how you can create a value of type `unit`:

```
ela> ()
```

Hint

Ela interactive console doesn't print out the value if this value is `unit` – therefore you will see nothing if you try to execute an expression above.

Unit is an immutable data type – you cannot change it in place like you do with an array (and there is nothing to change really). Because of this there is only one global instance of `unit`. If you create two values of type `unit` and try to test them for equality this will always evaluate to `true`:

```
ela>() == ()

True
```

Actually equality operators `(==)` and `(<>)` are the only two operators that are supported by `unit`. You are probably wondering what is the purpose of this data type which at the first glance cannot do anything useful.

The real application for `unit` is when there is no other meaningful value. Such as a function is evaluated only for the sake of side effect and doesn't have anything to return – that is the case when we can return `unit`.

It is important to understand here that unlike many other languages Ela doesn't have a notion of `null`. That is because Ela is a dynamically typed language and the *type tag* is not associated with variable but with the value itself. In other words it is not correct to say that in the code like `let u = ()` we have a variable u of type `unit`. In reality what we have here is a value of type `unit` which is bound to the name u. As a result you cannot declare a variable of type, say, `list` and assign it to a `null` like you do in C# or similar languages.

The `unit` type might seem as something similar to `null` and suitable as its replacement however `unit` is not `null` and is not here to serve the same purposes as `null`. You can use `unit` only when no other value is applicable. That is it. And it is a typical case for the functions with side effects when in C# you will have a procedure that returns `void` or accepts zero arguments. Ela doesn't have `null` and it doesn't need `null`. Also Ela doesn't have a lot of problems connected with `null` such as annoying *null reference* exceptions. If you used `null` in functions as a return value to denote that there is *nothing* we can return in a particular case please bear in mind: a much better way to do the same thing exists in Ela – *variants*. Don't worry yet, I will describe them later.

3.4. Syntax

Ela has indentation based syntax (also known as *two dimensional* or *layout based* syntax). The indentation rules in Ela are pretty close to the rules used in Haskell (which is of no surprise taken into account the similarities in syntax).

The first thing you should remember is that Ela really hates tab character. It is a syntax error to have even a single tab character in your code.

This was done for a reason – an ability to mix tabs and spaces in languages with indentation based syntax may lead to non-obvious syntax errors and it is much better to stick with either spaces or tabs and disallow usage of both. As soon as it is pretty hard to get rid of all spaces anyway tabs become a much better candidate for exclusion.

In general two dimensional syntax allows you to denote logical blocks in your code using indents. Indents play the same role as curly braces in C# or `begin...end` blocks in Pascal. It is important to understand than unlike some languages (such as Visual Basic or Ruby) Ela ignores line breaks and you still have a lot of freedom in your code formatting. Moreover just the regular formatting of code that you do anyway even in languages with a completely freeform syntax (like C#) will be enough for Ela to understand your code structure.

The rules are simple – if you have a single construct, such as a function declaration, all its content should be indented after the first line. For example, these are correct declarations:

```
let sum x y = x + y
let sum x y =
              x + y
let sum x y =
      x + y
```

But these are not:

```
let sum x y =
x + y
let sum x y =
    x + y
```

This is because Ela supports function definition by pattern matching and you can define several bodies for a single function like so:

```
let sum 0 0 = 0
    sum x y = x + y
```

And it is required to indent code in a function body after a function name.

A similar rule is applied when you use `et` keyword to declare mutually recursive functions. The following is a correct declaration:

```
let sum x y = x + y
 et div x y = x / y
```

And the following is correct too (but is less visual then the previous to my opinion):

```
let sum x y = x + y et
    div x y = x / y
```

But the following is not (because `et` is indented at the same level as `let`):

```
let sum x y = x + y
et div x y = x / y
```

The same for a pattern matching expression – match entries should be indented after the `match` keyword (and they all should be indented at the same level):

```
match [1,2,3] with
      x::xs = x
      [] = []
```

Indents are required to separate one match entry from another (as well as one function body from another). However if you need to write code in one line you can replace indents with semicolons like so:

```
match [1,2,3] with x::xs = x; [] = []
```

This may be useful in some cases, e.g. in interactive console as we have already discussed before.

Ela is an expression driven language and it has only two statements – global `let` bindings and module open directives. Ela allows you to have expressions directly in the global scope (but it might warn you if you are ignoring the value returned by one of them).

If you want to place several expressions in the top level one after another you can use a sequencing operator (pretty similar to the semicolon operator in C#). In Ela this is `($)`. I will tell you more about it in the chapter about imperative code in Ela. For now that is all you should know.

3.5. Comment convention

Ela has two types of comments – single line comments and multiple line comments. The convention for both of them is completely the same as in C#: single line comments starts with `//` and spans to the end of the line and multiple line comments are blocks enclosed in matching `/*` and `*/` pairs. Nested comments are not supported.

Chapter 4

Variables

4.1. A note on immutability

Usually a program in Ela consists of a sequence of variable declarations. You can declare a variable using `let` keyword like so:

```
ela>let foo = 42

ela>foo

42
```

I am using the term *variable* for the sake of simplicity; however it is not very accurate here. In reality we have declared a name `foo` and associated this name with a value `42` (which is as you already know a 32-bit integer). In functional languages this is called *binding*.

It is really not very correct to call `foo` a variable because it is not variable in the proper meaning of this word. Let's test that. In Ela `(<-)` is used as an assignment operator:

```
ela>foo <- 43

(1,1): Error ELA303: Unable to change a value bound to the name 'foo'.
```

I believe it is pretty self explanatory. You have already bound a name to a value – you cannot suddenly change your mind and redefine an already existing name.

Ela is a functional language. It does support mutating state like it was mentioned in the introductory chapter (we will have a separate conversation about it later) however that is not the way how you would normally program in Ela. Remember mathematics? Ela program is actually like a mathematical formula. First you say: let `foo` be `42`, but then, a couple of minutes later, you decide that it is better be `43`. Sorry, this is not how it works. If you really have reconsidered, why not to change your code? Or probably you need a different name for `43`?

```
ela>let bar = 43

ela>bar

43
```

I understand that inability to change a value of a variable, or better to say – the lack of variables in Ela – might seem as a very serious limitation for you, if you still think in an imperative style. But in Ela you don't write your programs as sequences of actions that change the state. One can even say that typical Ela programs are *stateless*. You probably wonder how it is possible to code anything useful without the mutating state. When a person from a functional world will be surprised that a program plagued with side effects can ever work correctly.

One of the great benefits of learning a functional language is that it will definitely change the way of your thinking – at least a little bit. Even if you continue to use imperative languages, you will look at your old code at a completely different angle. You will see that in a huge number of cases you don't have to fully rewrite your code to get rid of side effects.

A lot of tasks fit pretty well in this stateless functional programming model. One of them is something that we can call *transformation*. It is when you receive an input and need to apply a number of rules to it according to which this input gets transformed to a new structure. There is a whole family of applications that fall under this category. For example, compilers.

Of course there are tasks where mutating state is inevitable. And that is exactly the reason why Ela does support mutating state and is not a pure functional language like Haskell. But we will discuss it later. This chapter is about variables and immutability. So let's stay on topic.

As soon as you are not able to change a value bound to a name, Ela doesn't allow you to declare names that are not bound to something – in other words that are not initialized with a value:

```
ela>let foo2

(1,1): Error ELA310: Names cannot be declared without initialization.
```

That is perfectly reasonable because a name that is not associated with a value is completely useless.

4.2. Global and local bindings

You can use `let` directive to do both global and local bindings. We have declared only global names so far. I hope that you have familiarized yourself with the syntax already. All global bindings are not expressions and don't yield values.

Ela also allows you to declare local names. Moreover it support lexical scoping like C#, Haskell, OCaml, C and many other languages. In order to make a local name binding you should use `let...in` expression. Yes, you've heard right. It is an expression and can be used everywhere expressions are allowed:

```
let global = let local = sin x * 2
             in local / 2
```

Because `let...in` is an expression, it always yields a value, and a value that is returned is a result of evaluation of an expression that immediately follows `in` keyword – as you can see from the code sample above.

An equivalent code in C# will be as follows:

```
var global = default(int);

{
    var local = sin(x) * 2;
    global = local /2;
}
```

C# as a true C style language allows you declare lexical scopes using curly braces. You can do it any place you want with arbitrary nesting and you even don't have to declare variables – a code block enclosed in curly braces with no new variable declaration is a perfectly legal construct in C#.

Ela goes in a different way. You can probably agree with me that there is no much sense to start a lexical scope when you don't have an intention to create new variables. There is a reason why these scopes are called lexical! That is why the only way to create a new lexical scope in Ela is to declare a new name – like we did with `let` binding.

Names declared with `let...in` binding are visible only in expression that follows `in` keyword and not outside of it. Let's test it in interactive console:

```
ela>let y = let x = 42 in x

ela>y

42

ela>x

(1,1): Error ELA611: A name 'x' is unresolved. It is not declared
externally or its type has changed.
```

You can also nest `let...in` bindings at any level. Names, declared in parent bindings, will be visible in child bindings but not vice versa.

There is another important difference from C# here. Ela allows names in child scopes to shadow names in parent scopes. For example, the following code:

```
ela>let x = let x = 42 in x

ela>x

42
```

is completely legal in Ela but it's C# version won't even compile. However it will be valid in other C style languages such as C++.

4.3. One and another

But what if you want to declare several names at once? This is also possible using `et` keyword:

```
let result = let x = 42
              et y = 2
              in x / y
```

When you declare several *functions* using `et` keyword, these declarations become mutually recursive like in example:

```
let take x::xs = x :: skip xs
    take []    = []
 et skip _::xs = take xs
    skip []    = []
```

Here we reference `skip` function from `take` and `take` from `skip`.

Hint

Indenting rules for a chained name declaration using `et` keyword are simple. Keyword `et` should be either kept on the previous line (single line declarations are valid as well) you can move it to the next line but indent after `let` (or `where`) keyword.

However if you declare regular variables in the same manner they are *not* mutually recursive and the following code won't even compile:

```
ela>let x = y + 2 et y = 2

(1,9): Error ELA611: A name 'y' is unresolved. It is not declared
externally or its type has changed.
```

The reason for that is obvious. As you know Ela is a strict language therefore it does execute all expressions in order, unless you explicitly tell it not to do so. In the example above Ela will first try to calculate the value of x – but in order to do so it needs a value of y which is not yet initialized. As a result we have a dead loop. The situation is completely different with functions because the function code itself is not executed when a function gets bound to its name – therefore it is pretty safe to allow mutual recursion there.

Chaining declarations using `et` is allowed in both top level and local `let` bindings.

4.4. Where is my variable?

Sometimes declaration of local variables using `let` is not visual enough and tends to produce a not very readable code. Ela has another construct to do local bindings that you have already seen in the code samples before – a `where` binding. Let's see it in action:

```
let x = y
        where y = 42
```

This particular code sample is absolutely equivalent to the following:

```
let x = let y = 42
        in y
```

For those who came from C# or similar languages it might seem a little bit weird to refer to a name before it is declared. However it is up to you to decide which declaration – with `let` or with `where` – works better for you. I will probably peek the one with `where` as soon as it seems clearer to me and exactly follows the mathematical notation – the whole function body can be read as a single formula.

The `where` declaration allows you to do all the same things as `let` declaration. You can chain several declarations using `et`:

```
let z = x / y
        where x = 42 et y = 2
```

You can declare functions with `where` and use pattern matching. (I will tell you about these things later in the appropriate chapters).

In most cases `where` is just a syntax sugar for `let...in` however there are situations when `where` binding allows you to do things that otherwise will be impossible. We will see these unique applications for `where` in the chapter about pattern matching. For now you can just remember that even if you don't like `where` binding there are cases when you will have to use it anyway.

4.5. Naming rules

Yes, there are rules. When you are thinking to which name you want to bind some kind of a value you can choose any you like, but only if it complies the rules defined by Ela grammar. These rules are mostly the same as in C# but not exactly – and it is worth to spend a minute to memorize them.

An identifier in Ela can contain alphanumeric characters but it can start only from an alpha character or an underscore. It is no different from C# yet. The difference is that alternatively you can use a single quote character in your variable name. A name can't both start and end with a single quote character – otherwise it will be indistinguishable from the char literal. A name can start from, end with or contain a single quote somewhere in the middle. In other words – you can use a single quote character everywhere in the name, just make sure that it doesn't both start *and* end with a single quote.

Let's see at some example of valid Ela identifiers:

```
let o'neil = "O'Neil"

let 'foo = 42

let bar' = 12
```

The names `bar` and `bar'` are different names in Ela (like `bar` and `_bar` in C#) and the latter that ends with a quote is usually used for functions that does merely the same job as similar named functions but with some important difference – e.g. they have lazy, not eager semantics.

4.6. Name without a name

Believe me or not, but in Ela it is possible to use regular declaration syntax to do a thing that we can call an anonymous binding. What's that?

```
let _ = 2 + 2
```

You can't have a name that consists of just a single underscore and as a result by writing the code as above you don't bind a result of `2 + 2` calculation to any value. At the first glance it seems that this whole construct is redundant as soon as it doesn't do anything useful but in reality you will see similar declarations pretty frequently in Ela code. What is the purpose of them?

Let's take a look at another example:

```
let foo = 42

foo
```

```
let bar = 12
```

This is a valid code in Ela but when you try to compile it you receive the following message:

```
(3,1): Warning ELA404: Value returned by this expression is not used.
(3,1): Hint ELA501: Use 'let _ = ...' expression to disable warning, e.g.
'let_ = x'
```

I believe that the compiler hint unveils the purpose of the `let` binding with an underscore.

Ela does allow you to write expressions directly in the top level like statements. But expressions are called expressions because they yield a value. And when they are used like statements the returned value gets simply ignored. It is not a very good thing and that is a reason why compiler generates a warning as above.

In spite of the fact that our code sample with `foo` and `bar` is pretty useless, sometimes you might need to write code like this, e.g. when doing some calculations just for the sake of side effects. And `let _ = ...` construct is a way to tell a compiler that you are ignoring a value intentionally and not by accident.

4.7. Arguments

Arguments belong to a very special type of names which are not explicitly declared but can be used in our code in a manner similar to the regular names (and in a matter of fact are immutable as well).

Arguments in Ela are special names bound to parameters that are passed to Ela program from some external source. A good example are command line parameters. All arguments are clearly distinguishable from regular names and start with a dollar sign like so: `$arg`. When Ela compiler sees such a name it assumes that it is an argument and therefore doesn't have a declaration in the code. Ela is able to track arguments and if you try to execute code with a reference to an argument that haven't been provided an error is generated:

```
ela>$arg * 2

(1,1): Error ELA610: An argument 'arg' is unresolved. No argument with such name
 were provided.
```

Ela doesn't have any restrictions on the number of arguments that you can have and on their naming. When embedding Ela in your C# application you will be able to write code like so:

```
var res = Ela.Eval("$x + $y", new {x = 2, y=2});
```

But we will deal with this later, in the chapter about embedding Ela.

Ela Console also allows to use arguments and pass some values to Ela program from command line, but it simplifies the way how you can work with arguments a little bit and creates a single argument called `args` of type `tuple` that contains all the values passed from command line.

Hint

We will have a separate chapter about tuples. If you ever programmed in C# 4.0 or higher you are probably aware of this concept. However tuples in Ela are somewhat different from their implementation in C#. For now you can simply think about tuples as of read-only indexed arrays.

With Ela Console you can pass an argument to Ela code using `-arg` switch like so:

```
c:\>elac -arg "Hello, Ela!" -arg "Bye, Ela!
```

And this is how you can refer to the arguments passed:

```
ela>$args.[0]

Hello, Ela!

ela>$args.[1]

Bye, Ela!
```

Try to be careful here – if you expect an argument that wasn't passed and try to refer to it by its index you will catch a run time error:

```
ela>$args.[2]

(1,7): Error ELA805: Index '2' of type 'int' was out of range on object
'(Hello, Ela!,Bye, Ela!)' of type 'tuple'.
        in memory at line: 1, col: 7
```

But you will have a similar behavior even in a statically typed C# – so there shouldn't be anything new for you here.

4.8. Function declarations

I hope that you have already sorted out that functions in Ela are first class values. In fact there is a similar concept in C# called *lambdas*. When you are declaring a lambda in C# you are basically using the same syntax as for any variable declaration:

```
Func<int,int,int> fun = (x, y) => x + y;
```

That is because functions as first class values are in certain sense no different from any other values like, for example, integer numbers and strings. But in C# you have other types of functions that are not first class and require special declaration syntax. In Ela however all functions are first class values and as a result you can declare functions using regular `let` and `where` bindings – because you are simply constructing a value of a specific type and binding it to a name like in all other cases:

```
let sum = \x y -> x + y

let sum' x y = x + y //this one is equivalent to the one above
```

I have mentioned many times that Ela is a functional language and it's probably not a big surprise that functions are the most fundamental data type in Ela. And as soon as they are so fundamental they definitely deserve a separate chapter.

Chapter 5

Functions

5.1. Functional type

We have already acquainted with functions in previous chapters. Here I will summarize what we know about them so far and tell you about new cool things that you can do with functions.

Ela has its own convention to represent functional types. As soon as all type deduction logic is deferred until run-time Ela uses asterisk symbols instead of type names:

```
ela>let sum x y = x + y

ela>sum

sum:*->*->*
```

You see – when you print a name of a previously defined function in Ela Console this function gets serialized to a string representing the function type.

The notation `sum:*->*->*` is used to describe a function that accepts two parameters. The third asterisk stands for the return type. The word `sum` is a function name. If Ela doesn't know what the function name is it will replace the name with `<f>`:

```
ela>\x y -> x + y

<f>:*->*->*
```

If this notation to describe function type seems strange to you try to remember what we have learned about functions so far.

All functions in Ela accept just a single argument. If you need a multiple argument function you have to write a function that returns another function and so forth. The declarations above are just syntax sugar for such nested functions chain.

And that is what we see in a function type description in the form `*->*->*`:

- If you pass just a single argument you will get a new function as a result: `*->*`.
- If you pass two arguments you will get a saturated result.

Pretty simple, isn't it?

Now let's finally see what are the reasons, why functions are one of the most fundamental types in Ela.

5.2. Functions as first class values

You've heard this couple of times already. But what this concept of first class functions gives to developers except of the lack of special syntax for function declaration? Believe me in reality it unleashes a lot of programming power. This is just a small list of what you can do with it:

- Apply operations to functions just as to any other values
- Pass functions as arguments to other functions, yield them as return values
- Bind functions to variables, add them to lists, records, tuples, etc. as regular elements
- Create new functions on the fly by combining existing functions (which is called *function composition*).

Some modern programming languages – even not from the functional family – also utilize this concept of functions as first class values. Probably the most widespread language of such kind is JavaScript. C# as you know also has some limited support for the first class functions which is implemented through so called lambdas.

The main benefit of first class functions is that they build a basis for a functional programming paradigm. And one of the most important things in functional programming, something very closely related to first class functions, is *higher order functions*.

5.3. Higher order functions

Higher order functions are functions that operate with other functions – accepts them as parameters or yield them as return values. With higher order functions you can write a generic algorithm by moving a part of its logic outside and accepting it in the form of function arguments. Let's take a *fold left* function as an example.

In this chapter we will use Ela Console in multiple line mode, so lunch with a `-mt` switch and enter the following definition:

```
ela>let foldl f z x::xs = foldl f (f z x) xs
--->    foldl _ z []    = z;;
```

As you can see `foldl` accepts three arguments – a function, an initial value and a list. The first argument is a function for two arguments. What `foldl` does is iterating through the list and calling a given function on each element. As soon as the function that we pass to `foldl` accepts two arguments we also need a so called initial value (z) that is used to call this function for the first time.

OK, good. But at the first glance `foldl` doesn't do much. And that is true – it simply implements the idea of iterating through a list and calling a supplied function for each element, in other words – folding this function through the list. How this can be useful for us?

Let's say that we need to sum all elements of a list. This task sound a little bit more practical, does it? And the cool thing is that you don't have to code all the logic from scratch, you can use an existing `foldl` function:

```
ela>let sum = foldl (+) 0;;
```

```
ela>sum [1,2,3,4];;

10
```

Here we have created a new function by combining existing functions – `foldl` and `(+)` operator which is also a function as you remember.

Another example – you need a function that returns a maximum element from a list. No need to code the whole logic, `foldl` is eager to help:

```
ela>let maximum x::xs = foldl (\x y -> if x > y then x else y) x xs;;

ela>maximum [1,10,3,4,2,5,6];;

10
```

There is even a better way to accomplish this. Today we need to find a maximum element, tomorrow we will need a minimum element. As you can see implementation of a `minimum` function is almost similar to the maximum – we just need to apply a different operation to x and y from the anonymous function definition.

So let's build another higher order function and call it `elemBy`:

```
ela>let elemBy p x::xs = foldl (\x y -> if p x y then x else y) x xs;;
```

And now let's create a new version of a `maximum` function:

```
ela>let maximum = elemBy (>);;

ela>maximum [1,10,3,4,2,5,6];;

10
```

And a `minimum` function:

```
ela>let minimum = elemBy (<);;

ela>minimum [1,10,3,4,2,5,6];;

1
```

Pretty neat, isn't it?

But let's move forward. Now we have a different task – we need to reverse elements of a given list. It seems quite different from what we did before. But you know what? We still don't need to write an implementation from scratch:

```
ela>let reverse = foldl (flip (::)) [];;
```

And again – we were able to successfully accomplish the task just by combining existing functions.

Hint

> If you don't understand some of the code samples above – don't worry. Everything will be completely clear to by the end of this book. For now you should only understand the main concept of higher order functions.

Basically this is what makes a language functional – instead of coding the sequence of actions to operate with data you operate with functions, create new functions by combining existing functions. As a result instead of working with data and *thinking data* you think in terms of functions and what functions you need to combine to achieve your goal.

5.4. Tail recursion

We have briefly discussed this in introduction however it is worth to touch this moment one more time.

As you already know Ela doesn't have cycles such as `for`, `foreach` or `while` which you remember from C#. Instead one should use recursion. Of course recursion is not the most effective replacement for cycles. More importantly abuse of recursion in C# might cause such nasty things as stack overflow.

Most of functional languages try to deal with this problem by optimizing recursive calls away. Unfortunately it is not possible to do so in all the cases. However one particular scenario when such optimization in comparably straightforward is *tail recursion*.

As you remember tail recursion is when a function call to itself is the very last expression in the function body. A good example of this technique is a `foldl` function from the previous section. Let's take another example:

```
let length [] = 0
           x::xs = 1 + length xs
```

What about this function – is it tail recursive? The right answer is no. It might look like a tail recursive, but in fact the last operation in this function is not a call to itself but an addition. Operators in Ela are just functions and code `1 + length xs` is equivalent to `(+) 1 (length xs)` which makes it clear that we are not having a tail call here.

In many cases functions without tail recursion can be written with tail recursion. Let's optimize our `length` function:

```
let length n [] = n
           n x::xs = length (n + 1) xs
```

I am sure that you've got the trick. This is a pretty common way of turning non-tail recursive functions into functions with tail calls – you just promote the calculation that otherwise will be breaking our tail recursion into function arguments.

But now we have to specify an additional argument each time we call `length` function. And this argument will be always `0`. But I bet you already know how to deal with that – using currying and a local function:

```
let length = length' 0
             where length' n [] = n
                   length' n x::xs = 1 + length xs
```

You will see definitions in similar manner pretty frequently in Ela code.

5.5. Closures

All Ela functions with absolutely no exceptions are closures. We have touched this peculiarity already but I will summarize the idea anyway. As soon as in Ela you can declare functions anywhere – and that is the part of the obligatory to have first class functions – it is frequently pretty useful to refer to variables that are initialized outside of the function body, e.g. in the parent scope above the function definition.

Functions that are able to perform such a trick are called *closures*. It is important to understand that closures don't capture simply the variable value but the variable itself – you can think of it as of a possibility to refer directly to the parent scope variable within your function body. You should also be careful with it when operating with mutable data structures (like arrays or records with writable fields). All assignments will be immediately reflected in your closure function.

Sometimes such side effects may be useful in your programming but normally these are exceptional cases and in all other situations you should try to avoid them as soon as they render your code more error prone and more difficult to understand.

Closures are pretty useful even if you program in a pure functional way, with no side effects at all. In many cases you might need to fetch a function as a return value and this function will most probably perform some operations with the parameters of its parent function. In reality this is the only way how you can represent multiple parameter functions in Ela because all functions in Ela are *curried*.

5.6. Curried functions

You should be already aware that all Ela functions accept just a single parameter. If you want to implement a function that can operate with several parameters you can create a chain of nested functions like so:

```
ela>let sum3 = \x -> \y -> \z -> x + y + z

ela>sum3 2 3 4

9
```

Hint

As you can see the function that performs an actual calculation can refer to variables `x` and `y` declared in parent functions because it acts like a closure and captures the variables declared in the enclosing scopes.

Functions like `sum3` are called *curried* functions. As you can see a curried function creates a closure for each subsequent argument and if you need a function for three arguments you will have to create a chain of three closures as above, for four arguments – chain of four closures, etc. Also as you should remember there is a syntax sugar that simplifies declaration of such functions. Using this alternate syntactic construct the `sum3` function can be declared like so:

```
let sum3 = \x y z = x + y + z
```

Or even like so:

```
let sum3 x y z = x + y + z
```

Because function application operation is left associative you don't have to always use parenthesis when calling a function with several arguments, e.g. `((sum 2) 3) 4` is fully equivalent to `sum 2 3 4`. But, hey, you should know this already!

Curried functions are not here in order to make a language weird but because they unleash a very powerful programming technique – partial function application. As soon as all multiple argument functions are in fact just chains of nested functions you don't have to provide all arguments when calling such functions. For example calling a function `sum3` like so: `sum3 0` creates a new function for two arguments that sums these arguments with zero.

As you already learned functional programming is all about creating new functions on the fly by combining existing functions – and partial application is a tool that comes up really helpful here. You can implement a generic function and specialize it by fixing some of its arguments.

A good example of such technique is a `foldl` function defined in a previous section. Another example is a `zip` function. This function takes two lists and returns a single list filled with tuples that combine elements from both first and second lists. In other words if we have lists `[1,2,3]` and `['a','b','c']` as an input the resulting list will be `[(1,'a'),(2,'b'),(3,'c')]`.

How we can implement this function? As usual we can start with a more generic one:

```
ela>let zipWith z x::xs y::ys = z x y :: zipWith z xs ys
--->    zipWith _ _     _     = [];;
```

As you can see besides two lists `zipWith` also accepts a function as a very first argument and this function gets applied to elements of both first and second list when a resulting list is constructed. Using `zipWith` we can seamlessly create our target `zip` function:

```
ela>let zip = zipWith (\x y -> (x, y));;

ela>zip [1,2,3] ['a','b','c'];;

[(1,'a'),(2,'b'),(3,'c')]
```

A more straightforward example is a `take` function:

```
ela>let take 0 _     = []
--->    take n x::xs = x :: take (n - 1) xs
--->    take _ []    = [];;
```

What do we see here? A `take` function accepts an integer (that specifies a desired number of elements) and a list and returns a specified number of elements from this list. OK, let's assume that we have a pretty huge list and we need only first five entries from it, here is what we can do:

```
ela>let hugeList = [1..1000];;

ela>take 5 hugeList;;

[1,2,3,4,5]
```

It works however if you need to perform this operation frequently it is probably not very practical to always hard code the number of items that you want to fetch. As usual you can create a specific version of `take` function like so:

```
ela>let take5 = take 5;;

ela>take5 hugeList;;

[1,2,3,4,5]
```

Of course real life is not that simple and sometimes you might wish to "fix" an argument that doesn't come first. For example, we have a division function:

```
ela>let div x y = x / y
```

Now we need another function that divides all its arguments by two. But in order to reuse existing `div` function we have to partially apply it using only the second argument which seems impossible. Does it mean that currying is not going to help us here? It can – we simply have to rotate the arguments!

This can be done through a helper function like so:

```
ela>let flip f x y = f y x
```

And this is how we are going to use it:

```
ela>let div2 = flip div 2

ela>div2 10

5
```

Of course you don't have to define `flip` function everytime you need it. There is already a standard implementation of `flip` in Prelude which can work with functions that have any number of arguments (if this number is greater than one of course).

There are a lot of other things that you can do with currying. A curried function is just a function that accepts a single argument and returns another function. Moreover Ela is a dynamic language and it let you deal with types in any way you can think of. Remember we were talking about expressions that can have different type depending on values calculated in run-time?

Let's start from a simple example. We are going to import `Con` module that contains functions to work with standard input and output. It is pretty similar to the `System.Console` class in C#. Unlike C# however all functions in this module accept just a single argument when in C# you have overloaded versions of `Console.Write` method that allow you to pass an arbitrary number of arguments. Can we do a similar thing in Ela? The answer is yes:

```
ela>open Con;;

ela>let out x = write x $ out;;

ela>out "First " "Second " "Third ";;

First Second Third
```

Hint

See a `($)` operator? This one is pretty similar to a semicolon operator in C#. It is called a *sequencing operator*. It evaluates the left operand, ignores the return value and yields the value of a right operand.

A tricky question – how many arguments has an `out` function that is defined above? Well, probably not that tricky because you already know the answer.

The definition of `out` function is pretty straightforward: it accepts one argument, calls a `write` function which is equivalent to `Console.Write` and – now the important part – ignores the result of an application of `write` function (which is totally OK because it returns a useless unit) and yields itself as a return value. In two words – `out` is a function that returns itself. As a result – it looks like it can accept an infinite number of arguments. When in fact it accepts just one.

Let's take a more complex example:

```
ela>let sum' (x,y)   = \x' y' -> (x + x', y + y')
--->    sum' (x,y,z) = \x' y' z' -> (x + x', y + y', z + z');;
```

Out new function accepts a tuple and several other arguments that are added to tuple elements. The result is packed in a new tuple.

Now let's see what happens when we call this function with a pair:

```
ela>sum' (2, 3) 1 2;;

(3,5)
```

Good, it works. It did a "batch" sum of two values provided via tuple like so: `(2 + 1, 3 + 2)` which is obviously `(3, 5)`. But there is not magic here yet.

And now we call it with a triple:

```
ela>sum' (2, 3, 4) 1 2 3;;

(3,5,7)
```

That is more interesting. It looks like we are calling the same function with a different number of arguments. And it works. Now we have a batch addition of numbers in a triple like so: `(2 + 1, 3 + 2, 4 + 3)`.

And yet there is not magic. The `sum'` function simply checks – if we pass a pair to it than it returns a function for two arguments, if it is a triple – then we get a function for three arguments. In other words the return type of a function depends on the type of its input parameter.

That is a pretty neat trick and I am sure you can find a lot of pleasant applications for it in your code.

5.7. Functions without arguments

You probably noticed that we have only discussed functions that accept a single argument and always return something. Moreover I have stated at the very beginning that these are the only functions you can write in Ela. But how we deal with situations when we need a function that doesn't accept any arguments at all or doesn't return anything?

Let's say that you want to implement a function just for the sake of the side effect. This is perfectly valid in Ela as soon as Ela is not a pure functional language. This function accepts another function, a list and applies a given function to all elements of the list. This hypothetical function will also ignore a value that is returned by the supplied function. This is how we can implement it:

```
let each f x::xs = f x $ each f xs
    each _ []    = ()
```

I hope that this definition is clear to you.

As you can see we have defined this function by pattern matching as we did with `filter` in the first chapter. We pattern match against a linked list. The `($)` operator evaluates the first expression, ignores its return value and returns the result of an expression that follows. The second entry in our function `_ [] = ()` matches against an empty list and returns `unit` as soon as we have to return something but there is nothing else except `unit` that we can return.

And here is another example:

```
ela>let nonsense () = ();;
```

Here we have a `nonsense` function that accepts one argument of type `unit` and returns `unit`. This function is defined by pattern matching. As you remember from the first chapter with pattern matching you can use literal forms not only to create objects but to match against them as well. And that is exactly what we see here. A pattern `()` is used to match against a value of type `unit`. If you try to pass a value of other type a match will fail. This is how you can legally call this `nonsense` function:

```
ela>nonsense ();;

ela>let u = ();;

ela>nonsense u;;
```

And this is an illegal call:

```
ela>nonsense 0;;

(1,5): Error ELA809: Match failed. None of the patterns were successful.
        in memory.nonsense at line: 1, col: 5
        in memory at line: 1, col: 1
```

In other words we haven't tricked anyone here. We have created a function that accepts one argument. However by using `unit` we simply say that we don't really need anything meaningful with this argument and we are perfectly OK to accept a unit. What you see here is just a convention of representing functions without arguments. Of course you can do it differently like so:

```
ela>let nonsense2 _ = ();;
```

Here we used a *throw away* pattern which we also have seen before. It might seem like a good alternative but in reality it is not because you will be able to apply this function to a value of any type which is quite misleading:

```
ela>nonsense2 0;;

ela>nonsense2 "Hello, world!";;
```

5.8. Infix or prefix?

Normally functions are called using *prefix* form – that is by placing the function name before its arguments. A function call like `sum 2 2` or `take5 hugeList` are prefix forms. However sometimes it is more visual to use *infix* form – that is to place a function name between its arguments. This convention is typically used by operators however in Ela even regular functions can be called using infix form.

There are several cases when this possibility can be useful. Let's say that we have an `elem` function. This function accepts an element, a list and tests if a given element exists in a given list. This is how this function can be defined:

```
ela>let any f x::xs | f x  = true
--->                | else = any f xs
--->    any f []           = false;;

ela>let elem x = any (==x);;
```

> **Hint**
>
> Just a quick note – as you can see an `elem` function is defined through a more generic any function. The latter one accepts a predicate and a list and tests whether any element of a list satisfies a given predicate. The `elem` function simply defines this predicate as a function that performs a comparison of the provided value with a list element. As I've already mentioned above this is pretty typical style of programming in Ela.

OK, but what is so specific about this `elem` function? Mostly nothing. With the only exception that application of `elem` is probably easier to read when it is written in infix form:

```
ela>elem 5 hugeList;;

True

ela>42 `elem` hugeList;;

True
```

> **Hint**
>
> Please bear in mind that in order to call a function using infix form you have to enclose it in back apostrophes like in the example above.

That is not all. Let's say we don't need to scan our whole `hugeList` (which is huge) – we are perfectly OK with just first hundred elements from it. And we already have a `take` function for that:

```
ela>elem 42 (take 100 hugeList);;
```

It works but that is the code that I wouldn't call self explanatory. How we can enhance it?

```
ela>42 `elem` take 100 hugeList;;
```

Because regular function application binds tighter than function call in infix form we can use infix form to get rid of unnecessary parenthesis.

Functions are not the only entities that can be called using both prefix and infix forms. Operators that are applied using infix form by default can also be called in the same manner as functions:

```
ela>2 + 2;;

4

ela>(+) 2 2;;

4
```

Again it is important to remember that in order to apply an operator using prefix form you have to enclose it in parenthesis. Parenthesis are required because otherwise an expression `+ 2 2` will be interpreted differently. Ela will recognize the `2 2` as a function application (which binds tighter than any other operation) and a leading plus sign – as a partial application of an addition operator. And that is not what we are trying to do here.

In fact the very title of this section is a little bit misleading – infix and prefix forms are not the only possibilities. You can also apply both functions and operators using *postfix* form. Postfix means that an argument is placed before the function or operator name. This is how it might look:

```
ela>let isEven x = x % 2 == 0;;

ela>12 `isEven`;;

True

ela>13 `isEven`;;

False
```

Hint

We don't use parenthesis here because we only have a single argument. Moreover even when you use postfix form to call function the operation still remains to be left associative – because of that you can only specify a single argument. That is a limitation of a postfix form.

The same for operators:

```
ela>(2+);;

<f>:*->*

ela>let sum2 = (2+);;

ela>sum2 3;;
```

```
5
```

The support for the postfix form is really important when it comes to operators as soon as it unveils a very convenient way to partially apply operators. If you partially apply an operator using prefix form than the very first argument gets fixed. If you use a postfix form than the second argument is fixed:

```
ela>let div1 = 5-;;

ela>div1 3;;

2

ela>let div2 = -5;;

ela>div2 7;;

2
```

> **Hint**
> Of course if your operator accepts just a single argument than both prefix and postfix forms would lead to the same result – to the call saturation.

All these tricks work with regular functions as well however are a little bit more common with operators.

As a result, with all these variety of application forms and ability to switch between them, we finally come to a conclusion that there is no real difference between operators and functions. Or to say more precisely – operators *are* functions that use a different calling convention *by default*.

And that is really true for most of standard Ela operators. Which again leads us to another conclusion: if a difference is, well, basically non-existent why not to give users an ability to define their own operators?

5.8. Custom operators

Declaration of custom operators is no way different from regular function declaration – which is logical as soon as operators are just functions as we've seen before. There is one limitation however. In order to distinguish between operators and functions that have different default call convention Ela requires operators to have names composed from the following list of characters: `!%&*+-.:/<=>?@^|~`.

Let's see some examples of custom operators:

```
ela>let !! x y = x.[y];;

ela>hugeList !! 2;;

2

ela>(!!) hugeList 50;;

50
```

```
ela>let ^^ x = x * 2;;

ela>(122 ^^);;

244

ela>(^^ 42);;

84
```

It is also possible to define operators by pattern matching – exactly like functions.

But what about operator priority and associativity? Is it possible to specify them for custom operators?

Ela doesn't allow you to do that explicitely but uses a different approach (similar to such languages as OCaml and F#). In Ela operator priority and associativity is determined by the first symbol in operator name. This is a really simpler that it might sound.

If you declare an operator that starts with `(+)` it will have the same priority as addition, an operator with a leading `(*)` will have the same priority as multiplication and so forth. If you need a right associative operator with the same priority as construction operator you should start it with `(::)`. If you need a right associative operator with a high priority you should start it with `(<<)`.

The highest priority operators start with `(~)`, `(!)` or `(--)` and are usually used for unary operators.

5.9. Anonymous functions

I am really going to be short with these ones as we already have touched the basis before. Anonymous functions are functions that, well, are normally not bound to any name. A lot of modern languages support a concept of anonymous functions. For example in C# anonymous functions, or lambdas, are the only facility that can simulate functions as first class values and act as closures.

But this is not a case for Ela. It is important to understand that the only difference between a function created using lambda syntax and a regular function declaration is that the latter one is always bound to a specific name. You are also free to declare named functions using lambda syntax. For example the following declarations are completely equivalent:

```
let sum x y = x + y

let sum = \x y -> x + y
```

There is only the syntactic difference between the two. For Ela compiler those are just two ways to say the same thing. The latter one simply omits a couple of key strokes for you.

Anonymous functions are useful when you need to declare functions *in place* – to use them just once and then throw them away. However lambdas are not always the most concise way to declare a function. Sometimes partial application is much more readable and visual. Compare the two:

```
foldl (\x -> x * 2) 0 lst

foldl (*2) 0 lst
```

As you can see first sample with explicit lambda declaration is less readable than the partial application of multiplication operator. Moreover it is not even strictly necessary to use lambdas for *in place* function declaration. It is possible to use regular declaration syntax like so:

```
foldl (let doubleMe x = x * 2 in doubleMe) 0 lst
```

As you remember `let...in` always fetches a value by evaluating the expression that immediately follows the `in` clause. And that is the trick we use here. Of course this code sample is the most verbose among the three and probably the least practical however there is one difference between such declaration and a lambda expression.

Here you explicitly bind a function to a name therefore a compiler and afterward an execution environment know how the function supplied to `foldl` is called. And this will be used in exception stack trace if something goes wrong inside your `doubleMe` function which might simplify debugging in some cases.

But again you can rewrite the above example like so and it will be fully equivalent:

```
foldl (let doubleMe = \x -> x * 2 in doubleMe) 0 lst
```

There is only one difference except the whole anonymity thing between lambdas and regular functions. As you remember it is possible to declare a regular function by pattern matching and specify several function bodies for each set of patterns (that do align respectively with the function parameters). You can use pattern matching inside lambda declarations too however you can't define several bodies. This is the case when you should probably consider using regular declaration syntax.

5.10. Operations with functions

5.10.1. Application

Function application is most important and the most basic operation available for functions. In Ela it is defined as a simple juxtaposition of a function and its argument – in prefix, infix or postfix forms. There are two things however that you should remember about function application – I've mentioned them before but it is useful to repeat them one more time:

- Function application is left associative which means that function application in the form `(sum 2) 2` is always equivalent to `sum 2 2` and if you see an application like the one above you should understand that this is not a function call with two arguments but two subsequent function calls.

- Function application has one of the highest priorities among all other Ela operators. The only operation that has a priority even higher than function application is a member access operation (i.e. `rec.fieldName`). High priority means that in many cases you can omit extra parenthesis around application expression as soon as anyway it binds tighter than other operators, e.g. `nn :: (buildList s nn)` is fully equivalent to `nn :: buildList s nn`.

5.10.2. Pipe operators

Pipe operators are yet another way to apply a function to its arguments. Ela has two pipe operators – *backward pipe* operator (`<|`) and *forward pipe* operator (`|>`).

Let's see how to use them through an example. We have a function `doubleMe` that is defined like so:

```
ela>let doubleMe x = x + x;;
```

This is how you can call it using regular application syntax and pipe operators:

```
ela>doubleMe 2;;

4

ela>doubleMe <| 2;;

4

ela>2 |> doubleMe;;

4
```

As you can see backward pipe operator uses applicative order (just like the regular application) when forward pipe uses so called normal order.

> **Hint**
> Remember, pipe operators in Ela are just syntax and not functions. They are not defined in `Prelude`, you cannot override them and cannot use them as functions.

OK, but what's the catch? Why do we need these operators when they obviously seem to be redundant and do nothing more than the regular application does?

As you remember regular function application has one of the highest operation priorities. But pipes operators on the contrary have one of the lowest. And that is the catch. In many cases by using pipes operators you can omit a lot of parenthesis and make your code more visual.

This is a typical pattern when you have a chain of function calls. Let's see how it works:

```
ela>let when _ [] = []
--->    when f x::xs | f x  = x :: when f xs
--->                 | else = when f xs;;

ela>let project f x::xs = f x :: project f xs
--->    project _ []    = [];;
```

Here we have two functions – the first filters a given list using a given predicate and the second performs a list projection – applies a given function to all list elements. And this is a pretty typical scenario when you work with collections – fetch elements based on the certain condition and perform some transformations on the result list. This is how you can do it with forward pipe operator:

```
ela>hugeList |> when (>5) |> project (**2);;
```

```
[36,49,64,81,100,121,144,169,196,225,256,289,324,361,400,441,484,529,576,6
25,676,729,784,841,900,961,1024,1089,1156,1225,1296,1369,1444,1521,1600,16
81,1764,1849,1936,2025,2116...]
```

And with backward pipe operator:

```
ela>project (**2) <| when (>5) <| hugeList;;

[36,49,64,81,100,121,144,169,196,225,256,289,324,361,400,441,484,529,576,6
25,676,729,784,841,900,961,1024,1089,1156,1225,1296,1369,1444,1521,1600,16
81,1764,1849,1936,2025,2116...]
```

Forward pipe operator is left associative (which is the most natural behavior for an operator that applies functions from left to write) and backward pipe operator is right associative (which is again the most intuitive behavior as long as with backward pipe we are moving from right to left). Because of different operator associativity the two examples above are equivalent.

And now let's take a look at alternative construct written using regular application syntax:

```
ela> project (**2) (when (>5) hugeList);;
```

And we have only two functions here. Imagine the difference when you need to call, say, five functions in a row.

5.10.3. Function composition

As you can see pipe operators make your code much more readable in a lot of cases however if you require to call the same function chain frequently pipes are not really useful as soon as they require you to hard code the same chain of calls every time you need it. Function composition is a tool that is here to help you with such situations.

Ela has two function composition operators – *forward composition* (`>>`) and *backward composition* (`<<`). Composition is pretty similar to pipes in the sense of usage with the only difference – pipe operators are used to call functions and not to create functions, and composition is used to create functions and not to call them.

If one has to define an operator similar to, say, forward composition it might look like so:

```
let >> f g x = g (f x)
```

As you can see an implementation is pretty straightforward – we accept two functions and call the second one with the results of the first function application. Let's see it in action:

```
ela>let sqrt x = x * x;;

ela>let neg x = --x;;

ela>let div x = x / 2;;

ela>let c = sqrt >> neg >> div;;

ela>c 42;;

-882
```

Backward composition works in a similar manner but in reverse order:

```
ela>let c2 = div << neg << sqrt;;

ela>c2 42;;

-882
```

Now let's see how we can rewrite the sample from the previous section using composition operators:

```
ela>let qry = when (>5) >> project (**2);;

ela>qry hugeList;;

[36,49,64,81,100,121,144,169,196,225,256,289,324,361,400,441,484,529,576,6
25,676,729,784,841,900,961,1024,1089,1156,1225,1296,1369,1444,1521,1600,16
81,1764,1849,1936,2025,2116...]
```

Forward composition operator is left associative and backward composition operator is right associative.

Hint

Unlike pipes forward and backward composition are regular functions defined in `Prelude`.

Standard composition operators work only for functions that accept a single parameter. However if you have a more specific case you can always declare your own composition operator that will match your particular requirements.

5.11. Pure functions

Pure functions are yet another important concept in the world of functional programming. The concept is simple – you can call a function *pure* if it doesn't change state. A good example of a pure function is `sum`:

```
let sum x y = x + y
```

What can we say about this function?

First it doesn't have any assignments, doesn't change any global variables, doesn't try to change its arguments, etc. In other words – it doesn't affect the program state.

Then – which is closely connected with our first observation – this function is *deterministic*. In a case if you haven't heard this term before it simply means that if you call this function with the same arguments you will *always* get the same result. If one day you called it with arguments 2 and 2 and got 4, you can be absolutely sure that the next time you call with 2 and 2 you will receive exactly the same result. Just because 2 plus 2 is 4 – that is the basics of arithmetic that doesn't change from time to time.

These are very important things actually.

As soon as pure function is not affected by any other functions or values it is perfectly thread safe. Remember all this mess with synchronization logic in multithreaded C# applications? You don't need it with pure functions. At all. Also as soon as they always produce the same result for the same arguments you don't have to call them twice if the input didn't change.

The majority of functions that we write in Ela are pure. However Ela also allows you to create functions that are not very pure. So it is important not only to understand what makes a function pure, but what makes it impure as well.

A small question – is this function pure:

```
let foo x =
            let rec = {!value=0}
            in rec.value <- x
            $ rec.value * 2
```

The correct answer – yes, it is dumb, but pure. Why? This function has an assignment, mutable record and is pretty imperative. Yes, that is true. But being pure is no the same as being functional. You can write a very imperative function that uses mutable data structures, does a lot of assignments and other scary things but as long as it doesn't change anything outside of its scope (including function arguments) and produces the same result for the same arguments this function is pure.

And what about `Con.write` function (which is an equivalent of `System.Console.Write`)? Is it pure? No, it's not. This function always returns unit for any argument and therefore it formally satisfies one of the requirements. But it changes the external state – it writes to a console. Because of the same reason a function that deletes a file is not pure as well. Hey, it deletes a file!

So a pure function recipe is very simple. Don't do anything with the staff that doesn't directly belong to this function. Arguments don't belong to a function – they were passed from outside. Standard input/output doesn't belong to your function – and even if you read it, your function is not pure anymore. Even if your function doesn't delete a file but simply reads it – it's not pure. Because by reading file it still affects the external environment (such as placing a lock on this file).

5.12. Questions and answers

I bet you have a lot of questions. If we compare functions in Ela and C# then, in spite of all the cool features described in this chapter, you could probably notice the lack of many other things that are available for you in C# but wasn't even mentioned yet.

I will try to organize this section in Q&A style. So let's get started.

Q. What about optional parameters?

A. No, sorry. We don't have them.

The longer version of an answer. First of all optional parameters doesn't really fit well with curried functions. All our functions already accept just a single argument and all other arguments are, well, optional in certain sense. Second, you can actually do the things you normally do with optional parameters by using tricks like the one described in a section about currying.

Q. Is it possible to have `ref` parameters in Ela?

A. No. And one more time – no.

The longer version. Well if you are really asking this question then you probably need to read the previous section (called *Pure functions*) one more time. Anyway why do you need this thing? I always thought that using `ref` parameters is not a good style of programming even in C#. And Ela is a functional language. It is not pure, yes, but it still actively promotes the pure functional way of programming. Don't spread side effects, write pure functions! But I have to be frank – in reality you can do whatever you do with `ref` parameters in Ela also. But you will have to use records with mutable fields for that. There is a separate chapter about imperative programming in Ela and we will deal with this thing later.

Q. And what about `out` parameters?

A. No, don't even think about them.

The longer version. Well, there is no longer version. These `out` parameters seems equally evil as `ref` parameters to me.

Q. But how can I return multiple values from a function?

A. Ah, that is the right question. And here is the right answer – you surely can. With the help of *tuples*.

Chapter 6

Tuples and records

6.1. Tuples as a fundamental data type

Tuples are yet another fundamental data type in Ela. Tuples are grouped sequences of elements. You can access each element of a tuple using an indexer syntax and this operation is an efficient O(1) operation. In fact tuples in Ela can even remind you arrays in C#. But unlike arrays tuples are immutable – you cannot change them in place, append or remove items, etc.

Let's lunch Ela Console in a regular mode (without multiline support). This is how you can create a tuple in Ela:

```
ela>let t = (1,2,3)

ela>t

(1,2,3)
```

Tuple literal is very straightforward – you just list a sequence of elements separated by commas and enclose them in parentheses. However a single expression in parentheses like `(1)` doesn't create a tuple.

Parentheses in Ela are used as a grouping construct in the same manner as in C#. Therefore an expression `(1)` is completely equivalent to just `1` (but of course `1 + 2 * 3` is pretty different from `(1 + 2) * 3` exactly as in C#). If you want to create a tuple with just one element you can put a trailing comma like so:

```
ela>let t1 = (1,)

ela>t1

(1,)
```

But it is an error to leave a trailing comma if you already have a tuple with more than one element.

And yes – I almost forgot about this. You can have elements of different type in a single tuple.

```
ela>(1, 2.42, "Hello", 'c')

(1,2.42,"Hello",'c')
```

But there is nothing specific here really. As soon as Ela is a dynamic language you can have elements of different types even in a linked list.

6.2. Tuples and their usefulness

Tuples are really useful in certain cases. For example, with tuples you can initialize multiple variables with a single expression like so:

```
ela>let (x1,y1,z1) = (1,"Hello, world",42.12)

ela>x1

1

ela>y1

Hello, world

ela>z1

42.12
```

You can return multiple values from a function:

```
ela>let divMod x y = (x / y, x % y)

ela>divMod 43 12

(3,7)
```

In the example above I have created a function that accepts two arguments, divides first of them by second and returns the integral part of the division result and the remainder.

You can also use tuples to pass grouped arguments to a function:

```
ela>let sumTup (x,y) = x + y

ela>sumTup (1,3)

4
```

Here we have created a function that actually accepts just one argument. This function matches this argument and binds the first element of a tuple to x and the second to y. This function might look like as if it accepts two arguments not just one (for a C# programmer as long as C# uses such a similar convention for all function declarations and calls), however this is not the case in Ela:

```
ela>sumTup 1 4

(1,5): Error ELA809: Match failed. None of the patterns were successful.
        in memory.sumTup at line: 1, col: 5
        in memory at line: 1, col: 1
```

Hint

I understand that it can be quite misleading for the first time – all these functions look like they accept multiple arguments just because their syntax is really close to the one that is used to declare multiple argument functions in C# (or any C style language). But believe me – by the end of this book you will stare at functions in C# and think that they all accept just a single argument.

You can think that `sumTup` declaration is equivalent to the following:

```
ela>let sumTup2 t = t.[0] + t.[1]
```

However it is not. We can see why in a simple test:

```
ela>sumTup2 t

3

ela>sumTup t

(1,5): Error ELA809: Match failed. None of the patterns were successful.
        in memory.sumTup at line: 1, col: 5
        in memory at line: 1, col: 1
```

As you remember `t` is a tuple with three elements that we have declared before. The `sumTup2` function works without errors – it doesn't care how long our tuple is and will accept a tuple with three, thirty three or just one element. In the latter case it will fail with a run-time error while trying to obtain a non-existent element, in other cases it will simply ignore all tuple elements except of the first two. Sometimes this is not the behavior that we actually need.

And here comes our original `sumTup` function. As you can see it doesn't really want to accept a three element tuple. That is because a three element tuple doesn't match the pattern in the function declaration where we have explicitly specified that we want a pair of elements, not a triple and so on. This is how we can make this function work:

```
ela>let t2 = (12,24)

ela>sumTup t2

36
```

You can use this technique for deconstructing tuples not only when writing functions but also in any case when you need to obtain a particular element from a given tuple. `Prelude` already contains a bunch of useful functions that works with pairs (`fst`, used to obtain the first element, and `snd`, used to obtain the second element) and with triples (`fst3` and `snd3` respectively). This is how you can use them:

```
ela>(fst t2,snd t2)

(12,24)

ela>fst t

(0,0): Error ELA809: Match failed. None of the patterns were successful.
        in memory at line: 0, col: 0

ela>(fst3 t,snd3 t)

(1,2)
```

6.3. Tuples and their peculiarities

One interesting peculiarity of tuples is that unlike lists tuples are always flat. In other words you cannot create a tuple that has other tuples as its elements. If you need something like that you should probably stick with lists. With tuples the hierarchy will be always flattened:

```
ela>(1,(2,3,(4,5)),6)

(1,2,3,4,5,6)
```

That is the reason why it is fairly simple to write a custom operator for constructing tuples:

```
ela>let => x y = (x,y)

ela>1 => 2 => 3

(1,2,3)
```

Next, you can use the built-in `length` function to calculate the length of tuples:

```
ela>length (1,2)

2

ela>length (1,2,3)

3
```

Length calculation is an efficient operation with tuples – O(1).

And the final cool thing about tuples is that they support a lot of standard operators including comparison operators, arithmetic operators, bitwise, `succ` and `pred` functions and even concatenation operator. How all these things work with tuples? Let's take a look:

```
ela>(1,2) == (1,2)

True

ela>(1,2) < (3,4)

True

ela>succ (1,2)

(2,3)

ela>(12,5.5) + (1,4.32)

(13,9.82)

ela>(4,8) * (1,2)

(4,16)
```

As usualy there is no magic here. Tuples simply apply a requested operation to all of their elements. Want to sum two tuples? You will get a single tuple which elements are the result of addition of elements from the initial tuples. Want a tuple that is a successor to a given tuple? A `succ` function gets applied to each element of a tuple and a new tuple is constructed. Of course if try to perform some operation that is not supported by at least one element in tuple you will face with a run-time error:

```
ela>succ (1,"String",2,3)

(1,22): Error ELA826: A value 'String' of type 'string' doesn't support
operation 'succ'.
        in memory at line: 1, col: 22
```

6.4. Records

You can see that tuples are a very powerful data structure. But it has its own flaws also. First of all you can either access a tuple element by index which is not always convenient or you can use pattern matching to deconstruct a tuple which again is not always convenient especially if you need just a single element.

Writing code like so:

```
let (_,_,x) = t
```

is not something that you want to do frequently. And that is the reason why Ela has another data type – records.

Records are very much like tuples and in many cases they can be even treated like tuples (for example, you can deconstruct records using tuple pattern) however they do have their own important peculiarities.

I'll better put them in a list. What records can do (and tuples cannot):

- Access elements by names. With tuples you can only access an element by index when record allows you to give an alias to a particular element and to access this element using a given alias which might be more convenient in some cases.
- Mutability. Records are immutable by default but you can make them mutable if you need to do so (which cannot be done with tuples)
- Records can be cloned while tuples don't support cloning (and to be perfectly honest do not need cloning)

You might think that records are pretty close to anonymous types in C# – and in fact they are. But they are a little bit more flexible and powerful than anonymous types. As usual – a list of reasons:

- With records you can access elements both by name and by index or even use them like dictionaries
- You can return records from functions or pass them to functions when you might have obvious difficulties accessing a field of anonymous type outside of a function where it is declared
- You can pattern match records (using two different types of patterns)

- With records in Ela you can build both mutable and immutable data structures when anonymous types are only good for mutable.

6.5. Operations with records

Records in Ela are created like so:

```
ela>let r = {x=1,y=2}

ela>r

{x=1,y=2}
```

You can access elements of records using either their index or name. The latter one has a C style syntax:

```
ela>r.[1]

2

ela>r.y

2
```

You can also declare a record with field names that are not valid Ela identifiers (i.e. contain special characters or spaces):

```
ela>let style = {"text-decoration"="underline"}

ela>style

{text-decoration=underline}
```

In this case you will have to use indexer syntax to access a record element however you can still specify an element name through an indexer like so:

```
ela>r.["x"]

1

ela>style.["text-decoration"]

underline
```

I've mentioned before that you can use records as dictionaries however it is important to understand that records in Ela are not hash tables. They are associative arrays. All elements in records are stored in order – just like in tuples – and that is exactly the reason why you can pattern match a record using a tuple pattern. As a result we can use our previously defined `sumTup` function with a record:

```
ela>sumTup r

3
```

Of course there is a more specific pattern that is applied to records only. This pattern allows you to both match a record with the specified fields and bind values of these fields to local variables:

```
ela>let {x=x',y=y'} = r

ela>x'

1

ela>y'

2
```

First you have to specify the name of the field, followed by an equality operator and a name of a local variable that should be bound with the value of a field. In the example above we test if a given record contains fields x and y and bind the values of these fields to variables `x'` and `y'`. If a given records doesn't have any of the specified fields the match will fail. However if record contains other fields, not just x and y, it will still be a match. For example all the following patterns can be used to deconstruct a three element record:

```
ela>let xyz = {x=1,y=2,z=3}

ela>match xyz with {z=3} = true

True

ela>match xyz with {x=1,y=2} = true

True

ela>match xyz with {y=2,x=1} = true

True

ela>match xyz with {x=x,z=z} = x + z

4
```

There is also a syntax sugar that allows you to save up some key strokes when you want to bind field values to the variables of the same names. Instead of writing code like `{x=x,y=y}` you can simply write `{x,y}` – the latter syntax is fully equivalent to the first one.

The last useful operation with records that I am going to mention here is cloning. Cloning creates a shallow copy of a record and is implemented through a standard `clone` function defined in `Prelude`. Usage is simple:

```
ela>clone xyz

{x=1,y=2,z=3}
```

Actually records are the only standard Ela data type that supports cloning. Or better to say records are the only data type for which cloning makes sense.

As you understand it is pointless to clone an immutable data structure as soon as it doesn’t support changing in place and there is no reason why you might wish to create o copy of it. However records unlike all other built-in data types can be mutable.

6.6. Mutable records

One of the most important differences between records and tuples is that records can be mutable – you can change values of record fields in place without the need to create a new record. However this behavior is not supported by default. So if you try to change a value of our previously defined record `r` you will get the following error:

```
ela>r.x <- 2

memory(1,3): Error ELA816: Field 'x' of the record '{x=1,y=2}' is
immutable and cannot be changed.
        in memory at line: 1, col: 3
```

In order to make such an operation possible you have to explicitly declare a particular field as mutable using a `(!)` operator:

```
ela>let recMut = {!x=1,y=2}

ela>recMut.x <- 2

ela>recMut.y <- 3

memory(1,8): Error ELA816: Field 'y' of the record '{!x=2,y=2}' is
immutable and cannot be changed.
        in memory at line: 1, col: 8

ela>recMut

{!x=2,y=2}
```

And that is when cloning becomes useful:

```
ela>let recClone = clone recMut

ela>recMut.x <- 1

ela>recMut

{!x=1,y=2}

ela>recClone

{!x=2,y=2}
```

Chapter 7

Variants

7.1. Algebraic data type

In a type theory *algebraic data type* is a type combined from a set of values so that each of these values is a separate data type. Or, using a more accurate definition, an "algebraic data type may be viewed as an *abstract type* that is declared to be *isomorphic* to a (*k*-ary) product or sum type with named components"[2].

Most of object-oriented and imperative languages don't have a notion of algebraic data type however they might have other data structures that resemble a certain similiarity with algebraic types. In C# such data structure is called *enumeration*.

Imagine that we have a company that cells several products. We have a wide distribution network but range of our goods is somewhat limited. As a result we can easily present all of them through a single enumeration like so:

```
public enum Product
{
    Cellphone,
    Laptop
}
```

Our task is to write an application that monitors our sales and maintains stats with a number of cellphone and laptops sold each day.

At a first glance using an enumeration to represent a product type seems like a good idea but then we suddenly discover a number of pretty serious limitations that start to play a role of a "stopping moment" in our development.

One of the most impacting is that we can't associate any additional data with the product type such as technical description. Our first intention is to implement a custom data structure called `Product`, that will be used to describe a product type.

It seems like the way to go but when we start to design our data structure we suddenly find outselves in another architectural dead end. The products are different and they need a different set of attributes.

---

[2] *The Essense of ML Type Inference*, Francois Pottier and Didier Remy, p. 454, *Advanced Topics in Types and programming languages*, MIT, 2005

OK, I am probably over-dramatizing here. All we need here is to use one of the core object oriented programming features called *inheritance*. So we end up with creating a base class `Product` and its derivatives called `Cellphone` and `Laptop`:

```
public abstract class Product
{

}

public sealed class Cellphone : Product
{
    bool TouchScreen { get; set; }
}

public sealed class Laptop : Product
{
    double ScreenSize { get; set; }
}
```

Don't try to analyze the set of attributes which I have chosen for these classes – this is just a simplified example.

OK, we did a good job. But there is still one important requirement that we haven't taken care of. Because of some strange reason our company can sell only laptops and cellphones and nothing else than that. We have a life time guarantee that our range of products is not going to change. And we need to make sure that no one except of `Laptop` and `Cellphone` will be able to inherit from our `Product` class.

After some meditation we come up with the following:

```
public abstract class Product
{
    private Product() { }

    public sealed class CellphoneProduct : Product
    {
        public bool TouchScreen { get; set; }
    }

    public sealed class LaptopProduct : Product
    {
        public double ScreenSize { get; set; }
    }

    public static LaptopProduct Laptop(double screenSize)
    {
        return new LaptopProduct { ScreenSize = screenSize };
    }

    public static CellphoneProduct Cellphone(bool touchScreen)
    {
        return new CellphoneProduct { TouchScreen = touchScreen };
    }
}
```

As you can see I have also created special construction functions – let's call them *constructors* – to make our new data type a little bit more usable.

Now take a closer look at the definition of `Product` type. It is probably not the typical way how you design classes but at the same time we only use the basics of OOP and some of the features available for nested classes in C# (such as an ability to refer to private members of the parent class). Take your time, there is no rush. This `Product` type is really important for us. Let's see how we are going to use it:

```
var l = Product.Laptop(14.2);

if (l is Product.LaptopProduct) {
    ...
}
else if (c is Product.CellphoneProduct) {
    ...
}
```

I am so specific about this `Product` class because we have just created an algebraic data type in C#. Yes, truly. We have implemented all the major properties of it. We have a `Product` type that can be either `Cellphone` or `Laptop` and nothing else than that. This type is *a sum of products*.

Let's double-check if have really covered all the properties of algebraic data types:

- When compared with object-oriented class hierarchy algebraic data types support only one level of inheritance – that is when you can have cellphones inherited from products but you can't have smartphones inherited from cellphones. And we've done that – our derivatives are marked with the `sealed` keyword and the inheritance chain cannot be continued.

- Algebraic data type define a finite set of productions. You can have only products of type `Cellphone` and `Laptop` – and no products of other type. And we have fullfiled this requirement as well by making a constructor of class `Product` private.

So yes, we have really managed to create a true algebraic data type in C#.

7.2. Limitations of algebraic data types

Of course C# doesn't have what we can call a first class support for algebraic data types and we had to write quite a lot of code to simulate one. Other programming languages that support algebraic data types directly allow to do completely the same thing in a much more concise manner. For example this is how a definition of `Product` type looks in F#:

```
type Product =
    | Laptop of double
    | Cellphone of bool

let l = Laptop 14.2

let res = match l with
          | Laptop _    -> "We have a laptop"
          | Cellphone _ -> "We have a cellphone"
```

As you can see you can be much more expressive with a functional language that supports algebraic data types.

Of course the real life is not that simple. And in many cases a data structure that allows us to represent only a fixed set of productions might fail to fulfill the project requirements. We have used an imaginary company in our sample that can only sell laptops and cellphones when most of real companies don't have such strict constraints and might change their range of products almost any time.

The fact that algebraic data types are closed and inextensible might be their benefit and a serious downfall at the same – depending on what you actually want to express with them. It is a benefit because when you know that products can be either laptops or cellphones with no other alternatives you can build up the logic that is responsible for data analysis accordingly.

Let's take a look at the example in F# above. We have a match expression with just two entries. This match is exhaustive because it takes into account all the possible invariants. There is no way to break this code by supplying an unwanted `Product` derivative, because we can't extend `Product` from "outside", we always know that this code is one hundred per cent complete and exhaustive. If you try to manually add a new constructor to `Product` type like so:

```
type Product =
    | Laptop of double
    | Cellphone of bool
    | Monitor of bool
```

and compile your code, F# compiler will immediately track the potential problem with your `match` expression and generate an appropriate warning:

```
warning FS0025: Incomplete pattern matches on this expression. For
example, the value 'Monitor (_)' may indicate a case not covered by the
pattern(s).
```

Hint

In fact you will have a similar problem even in C# with our simulated algebraic data type – but compiler won't be able to warn you about it.

These are very good properties that make your code, well, more predictable, but sometimes we simply need extensible data structures. In computer science an inability to extend an algebraic data types is called an *expression problem*. As Philip Wadler said, "The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code"[3].

And unfortunately because of the way how algebraic data types are organized they are the not right tool for such cases.

7.3. Polymorphic variants

Luckily there are languages (or better to say – language designers) that just can't live with limitations. The language that I mean here is OCaml, which belongs to the same family as F#. OCaml supports classical algebraic data types but besides them it also provides a feature that is called *polymorphic variants*.

---

[3] *The Expression Problem*, Philip Wadler, 1998. Source: http://www.daimi.au.dk/~madst/tool/papers/expression.txt

Polymorphic variants are a rethinking of an algebraic data type concept that tries to solve an expression problem. And it is a very bold rethinking.

With polymorphic variants we no longer have to declare algebraic data types. Instead we claim that there is a single algebraic type in the world and all our constructors are its productions. As a result these productions are declared on the fly when you simply reference them within your code.

This is how our initial code sample in F# can be rewritten in OCaml:

```
let l = `Laptop 14.2;;

let res = match l with
          | `Laptop _    -> "We have a laptop"
          | `Cellphone _ -> "We have a cellphone";;
```

As you can see it is really very close to F# version but there is no declaration of a `Product` type and constructor names start with a back apostrophe.

First of all this brave new concept really does the job. We now have an extensible data structure. If tomorrow you need to add monitors to the range of our products – they are already added as soon as they are in your head. And you are free to write code like so:

```
let res = match l with
          | `Laptop _    -> "We have a laptop"
          | `Cellphone _ -> "We have a cellphone"
          | `Monitor _   -> "We have a monitor";;
```

Of course it immediately destroys the requirement for the pattern matching to be exhaustive. As long as your data type is mostly in your head and not in the code a compiler doesn't have any tools to check whether a match is exhaustive or not. In reality it means that all matches against polymorphic variants are always inexhaustive, there is always a risk that we will face with a constructor that we even haven't heard of. And to keep our code safe from unpredictable errors we better add a default entry to each `match` construct that will be used in a case when we find a constructor that wasn't in our head when we were writing this match.

As you can guess this behavior demonstrates an obvious problem of polymorphic variants. If regular algebraic data types are inextensible, than polymorphic variants are too extensible with absolutely no control of their extensibility.

And it sounds like a noticeable problem. OCaml is a very strict statically typed language and polymorphic variants introduce a feature that is very close to dynamic typing in its nature. I would even say that this feature seems like a little bit misplaced if we speak about OCaml. It's not bad itself and may be really useful and, well, if we have a dynamic language, where polymorphic variants could fit perfectly, as soon as a language is already dynamic and it can't be messed up with yet another dynamic feature...

Wait. Dynamic language? Like the one we are talking here about?

7.4. Variants in Ela

You know what – you can actually forget everything I told you in the previous sections because variants in Ela are much simpler than that. You don't have to know about algebraic data types, expression problems, polymorphic variants.

Let's put it strait.

Sometimes you might need to *tag* values. If you are programmed in C# it offers a similar yet pretty limited infrastructure called *nullable types*. Imagine that you write a function that might return either an integer value or nothing. How to return this "nothing"? Of course you can generate an exception but in many cases it is not a good idea as soon as we don't have any *exceptional situation* here – we just have nothing to return.

That is a place where nullable types comes very handy. With nullable types you simply create a structure that wraps your actual return value and a boolen flag that indicates whether this value was trully initialized. Let's implement a simple function in C# that accepts a single integer argument, tests whether this argument is odd and if it is returns the remainder of its division by 2, otherwise – returns "nothing":

```
int? getOdd(int x) {
  var r = x % 2;

  if (r > 0)
    return r;
  else
    return null;
}
```

Ela doesn't have nullable types. But Ela has something that might be a little bit more flexible. This is how this `getOdd` function will be implemented in Ela:

```
let getOdd x | r > 0 = `Odd r
             | else  = `Even
             where r = x % 2
```

In Ela you can also create a wrapping structure that can be used to attach any custom tag to a value if you want to associate some additional data with this value that can be used in further analysis. You can also attach a tag to `unit` (in a case if there is no meaningful value for the particular situation). It is done like so: `` `None () `` or even shorter like so: `` `None `` – these two expressions are equivalent.

And that is what we have done in the example above – for an odd value we attach a tag `Odd` to the remainder and for the even value (as soon as there is no remainder) we just return a plain tag `Even`.

> **Hint**
> Please bear in mind that variant tags always start from a *back apostrophe*. That is the place where Ela shares the same syntax with polymorphic variants from OCaml.

As usual you can pattern match agains variants using their literal syntax:

```
ela>let getOdd x | r > 0 = `Odd r | else = `Even where r = x % 2
```

```
ela>match getOdd 13 with `Odd x = x; `Even = 0

1
```

Tags can be attached to values of any types – it can be integers, tuples, lists, records – whatever you can think of. When pattern matching you should specify a tag name followed by pattern that is used to match against a value to which this tag is attached. Function `getOdd` uses a simple integer literal pattern but it can be head/tail pattern, list pattern, tuple pattern, etc.

Also there are two standard functions that can be with variants in `Prelude`. The first is called `tag`, accepts a variant a returns its tag as a string:

```
ela>let v = `Some 42

ela>tag v == "Some"

True
```

The second is `untag` function and basically "unwraps" a value packed in a variant:

```
ela>untag v == 42

True
```

Polymorphic variants are used very widely in Ela. In many cases they are a much better alternative to exceptions (which are also supported in Ela but used less extensively than in languages like C#). Moreover variants are actually one of the major ways how you can create your own data types in Ela:

```
ela>let cellp = `Cellphone { Model="E75", Maker="Nokia", Color="Black" }

ela>if cellp is (`Cellphone {Model="E75"}) then "This is a E75
smartphone!" else "Don't know what is is."

This is a E75 smartphone!
```

As soon as variants fully support pattern matching even exceptions in Ela are based on variants. Also a lot of standard functions use `` `Some `` and `` `None `` tags in their return values to indicate whether a value is actually there.

Chapter 8

Lazy evaluation

8.1. Eager and lazy

Lazy evaluation is an evaluation strategy that defers evulation of an expression until its value is actually needed. In many cases it may be seen as an optimization technique that ensures that all evaluations are perform on demand only.

There are programming languages (like Haskell or Clean) where lazy evaluation is an evaluation strategy that is used by default.

Lazy evaluation is closely connected with language purity – with the lack of side effects. This connection comes from the fact that with laziness there is no strict order in which your code is evaluated. That is something that might cause certain difficulties in understanding the logic flow if you still thinking in an imperative style developed through programming in languages with eager evaluation (such as C#).

The key thing here is that pure functional language don't even try to express the logic flow, the sequence of computations and other imperative things. In a pure language you have a set of declarations – such as pure functions declarations – which don't depend from each other, don't change global state or use any external resources and therefore can be evaluated in any order.

That is important. When your code doesn't have a mutatitive state the order of evaluation doesn't matter. Imagine that you have a simple code in C#:

```
var x = Math.Sin(8765.12) * 2;

var y = Math.Tan(233) / 3;

var result = y + x;
```

Expressions that are used to initiliaze variables x and y are pure. Calculation of sinus and tangent are functions without any side effects – they return the same result for the same argument all the time. That is true for multiplication and deletion as well. In C# these expression are evaluated in order – first we calculate the value of x, then – the value of y. However there is absolutely no difference in which particular order these particular expressions will be evaluated. You can calculate y before x – and the result will be the same. In fact in a pure functional language y will be indeed calculated before x – because addition operation is usually left associative and y is the first operand.

With all these precious knowledge you can now look at the small code sample above differently. There is no order. There is a set of declarations. And it doesn't matter when exactly initialization blocks in these declarations are evaluated. Simply because it doesn't change anything.

8.2. Explicit and implicit

But as you know C# is not a pure functional language. So as Ela and many other languages from the functional family. As long as you allow certain impurity in your code and therefore make possible for the expressions to depend from each other, an implicit "by default" laziness is definitely not the way to go. Lazy code with side effects is a nightmare for a programmer.

Imagine that you have the following code:

```
let x = 2

let y = x + 2

x = 3 //Not a valid Ela code

let z = y
```

If this code is executed in a strict manner than the value of `z` is 4. But if this code is executed in a lazy manner than the value of `z` is 5 because `x + 2` gets evaluated after the value of `x` is changed. It might be pretty frustrating – moreover it makes programming a hell of a task. Therefore lazy evaluation is not supported by imperative languages with side effects.

However even in impure languages you can writen crystal clean and pure code. You can easily do it in C#. And you can surely do it in Ela.

As long as you still can write pure code simply dropping lazy evaluation feature doesn't seem like a right decision. (Moreover without it it will be impossible to do a lot of things such as to represent infinite data structures – but we will talk about it later). But it is absolutely clear that implicit lazy evaluation (like in Haskell) is not acceptable either.

The solution is obvious – we need an explicit way to mark certain block of code as the one that should be evaluated in a lazy manner.

Ela has a special data type – *thunk* – that provides a support for lazy evaluation.

8.3. Thunks

Thunks allow you to do deferred evaluation. With this data type you mark a certain expression in your code as the one that should be evaluated in a lazy manner – not immediately after it is declared but only when you actually need its value (e.g. to perform some kind of an operation with it).

This is how you can initialize a thunk in Ela:

```
ela>open Con

ela>let thunk = (& let temp = 2 + 2 in writen temp $ temp)
```

As you can see we have an expression that initializes a local variable with a result of an addition of two numbers, prints this result to the output and returns it. But when you copy the code above to Ela Console you see nothing. That is because an expression in `(& ... )` braces is not evaluated. Instead an instance of a special data type is created that wraps this expression and `thunk` variable is initialized with an instance of this data type.

Hint

Using an impure function such as `writen` along with lazy evaluation is normally not a good idea; so don't think of an example above as of a recommendation of what you should actually do in your code. We are dumping temporary variable to console just in debugging purposes – in order to track a `thunk` calculation.

Now we know that when Ela will finally calculate our lazy expression a console output in it will be triggered and signal us that calculation is finally done.

Let's see what happens if we pass `thunk` variable to the function as follows:

```
ela>let isThunk x = if x is lazy then "Yep, it's lazy!" else "No, it's not!"

ela>isThunk thunk

Yep, it's lazy!
```

OK, it wasn't evaluated. Why? Because type check that we have performed in the `isThunk` function doesn't require an expression to be evaluated – and as the result it wasn't.

Let's try another way:

```
ela>thunk * 2

4
8
```

And now we have what we needed. We tried to multiply the value of `thunk` by two, however in order to do so we obviosly need to calculate it. And that is the thing that was done here as you can see by the console output.

But what is going to happen if we try to use `thunk` in another calculation?

```
ela>thunk / 2

2
```

It is not calculated anymore! That is because it already was evaluated and its value was effectively remembered in order to prevent unneccessary calculations in future. The rule is pretty simple – every thunk is calculated just once and after that it becomes just an ordinary value:

```
ela>isThunk thunk

No, it's not!

ela>thunk is ?int

True
```

8.4. Time to be lazy

Thunks may seem as an optimization and in many cases they can be used in such a way however they may even degrade your code performance. That is because thunks do generate additional overhead – for their initialization, operation and so forth. In Ela this overhead is not too high and you won't notice it in most cases – so do not afraid to use thunks wherever you need them. However in certain cases you might want to evaluate whether thunks are really needed. If you already have a pure expression and it truly doesn't matter when this expression gets evaluated evaluating it right away may be a good choice – especially if you need a result of this evaluation just once.

Thunks are really useful in the case when, depending on your logic, some values may not be evaluated at all.

Another unique application of thunks are lazy (and infinite as a particular case of laziness) data structures which can be implemented with thunks and, for example, linked lists.

Chapter 9

Lists

9.1. Lists as a fundamental data type

Yes, yet another fundamental data type. And that is true because when programming in Ela you will use lists all the time. We have already acquainted with the list through multiple code samples – and it was almost unintentionally because it is merely impossible to demonstrate a lot of things without lists.

Lists in Ela are *single linked immutable lists*. They are not different from other implementations of immutable lists but in a case if you haven't heard about lists before I am going to spend a minute or two just to give you the basic idea.

In short single linked list is an ordered sequence of elements where each element holds a reference to the next element. The very last item in a list points to a thing that is called a *nil list*, which is nothing more than an empty list.

It is important to understand that list is a recursive data structure. All elements of the linked list are also lists.

Basically a single linked list itself can be described as a wrapper for a value and a reference to another list. This is how a minimalistic single linked list can be implemented in C#:

```
public class LinkedList<T>
{
  public static readonly LinkedList Nil = new LinkedList();

  private LinkedList()
  {
  }

  public LinkedList(T value, LinkedList next)
  {
    Value = value;
    Next = next;
  }

  public LinkedList Next { get; private set; }

  public T Value { get; private set; }
}
```

That is really it. As soon as list is an immutable data structure an empty *nil* list is represented as a *global* value. You don't have to create it "on demand" as soon as it never gets changed and therefore can be initialized just once in a life time and then used everywhere. As a result when you create an empty list in Ela, nothing actually gets created – you just reference this special global *nil* variable.

9.2. Operations with lists

Creating an empty list in Ela is simple:

```
let lst = []
```

A `[]` is a special literal form that is used to denote an empty list.

If you need to create a list with just one element – e.g. an integer `42` – you basically have to initialize this simple linked list structure and to pass `42` as its value and a `nil` list as an element sitting nearby. This is how you can do it in C# using the `LinkedList` class that we've just created:

```
var lst = new LinkedList(42, LinkedList.Nil);
```

The same for the list with two elements:

```
var lst = new LinkedList(43, new LinkedList(42, LinkedList.Nil));
```

But as you can see it looks quite cumbersome. That is why Ela has a special operator `(::)` that can be used to construct lists. Let's try it in console (we are going to use single line mode this time):

```
>let lst = 43::(42::[])

[43,42]
```

An operator `(::)` is called a *construction operator* and it is right associative. It means that you can omit parenthesis from the previous sample and get exactly the same result:

```
ela>let lst2 = 43::42::[]

[43,42]
```

There is also a syntax sugar for this operator – a special literal form of the list. The code above can also be written like so:

```
let lst2 = [43,42]
```

Another useful operator that can be used with lists is a *concatenation operator*. This is how you can use it:

```
ela>lst ++ lst2

[43,42,43,42]
```

A ++ operator can be used to concatenate two lists, you can't use it to append an element to a list. Actually because of the way how linked lists are organized there is no direct way to append an element to the end of the list. But if you desperately need it there is a way to do that of course:

```
ela>lst ++ [41]

[43,42,41]
```

Hint

Please bear in mind that this not a very efficient operation – not only in Ela but in all languages that support lists and list concatenation.

Here we actually create a new list that contains a single element, number `41`, and then concatenate the two lists.

Standard Prelude defines several functions that can be used with lists.

First of all these are handy `head` and `tail` functions that returns a head and a tail of given list respectively:

```
ela>head [1,2,3]

1

ela>tail [1,2,3]

[2,3]
```

If you call head with an empty list an exception is raised. Your program will also fail with an exception if you try to obtain a tail of nil list. In order to make sure that your list is not empty an `isNil` function can be used:

```
ela>isNil [1,2,3]

False

ela>isNil []

True
```

Another useful function is a `length` function that you've already seen in the section about strings:

```
ela>length [1,2,3,4,5]

5
```

Standard `Prelude` contains only very basic functions that can be used with lists but you can find plenty of others in `Core` module that is also the part of Ela standard library. (There will be a separate chapter about Ela standard library in the end of this book).

You have already acquainted with some of them. These are `filter`, from which we have started to learn Ela, `fold`, `reverse`, etc. All these functions basically iterate through the list elements and create a new list either in reverse order or based on condition or using supplied function that is folded through the list elements.

9.3. Iterating lists

As soon as every list element holds a reference to the next element lists come up very handy when you need to iterate over their elements. In other words each item in a list basically know what its neigbor is and it renders looping through the list as a pretty trivial task.

However Ela is a functional language and it doesn't even have a concept of iterators from such languages as C#. In Ela one should use recursion instead of cycles. And that is again the place where *pattern matching* becomes really helpful. As you pattern matching allows us to deconstruct an element using its literal syntax which is in a case of linked lists a construction operator `(::)`. Here is a simple example of how you can iterate over list elements:

```
ela>let iter f x::xs = f x $ iter f xs; _ [] = ()
```

This function is basically an alternative to imperative cycle and you can use it like so (the following example prints all list elements to console):

```
ela>open Con

ela>iter writen [1,2,3,4,5]

1
2
3
4
5
```

You've already seen a similar trick before. Lists and pattern matching are widely used in Ela code. As you remember a pattern in the form `x::xs` matches against a list that has at least one element. If a list really contains just a single item `xs` gets bound to the `nil`. Usually in expressions like the one above we call `x` a *head* of the list and `xs` a list *tail*.

The list deconstruction pattern can contain an arbitrary number of elements. You can use both variables and actual values in it, e.g. the following patterns are perfectly legal: `1::2::xs`, `x::(2,3)::xs`, `x::(y::ys)::xs`, `x::y::[]`. I will explain how this thing works with more detail in the chapter about pattern matching.

The last pattern `x::y::[]` is used to match a list that has *exactly* two elements. We basically say here that we want a list tail to match a nil list. It is useful when you want to ensure that a given list has an exact number of items. As with construction syntax there is a more readable form to write this: `[x,y]` which is fully equivavlent to `x::y::[]`.

Here is another example of how you can write a function that iterates over list elements using recursion and pattern matching:

```
ela>let map f x::xs = f x :: map f xs; map _ [] = []

ela>map (*2) [1,2,3,4,5]

[2,4,6,8,10]
```

The function above is one of the most commonly used higher order functions (along with `fold` which we know already). This function accepts another function and a list and generates a new list by calling a supplied function on each element. Here we are using the `map` function to double all the list elements.

9.4. Indexing lists

Another useful operation that is available for lists is an ability to obtain a list element by its index. This is done by using indexing syntax like so:

```
ela>let newList = [0,1,2,3,4]

ela>newList.[2]
```

As you can see the code is exactly the same as with tuples. But you should remember that accessing list elements by index is not a very efficient operation. We have to iterate through the whole list and count each iteration – that is really the only way to do it. If you need to access elements by index frequently you might consider to use a different data structure instead – e.g. *tuple* which has a constant access time.

9.5. Lazy lists

One of the coolest properties of lists is that lists can be lazy. We have already acquainted with the concept of laziness in the previous chapter and basically there is nothing new to learn here.

Lazy list is a list that is created on demand, only when you need a value of a particular element. For example you may create a list of twenty elements but if you will finally need only first five of them – creation of the rest is useless. But it is not a problem with a lazy list which gets constructed when you actually try to acquire its items.

But that's not all. You can also use linked lists to represent infinite lists – lists that basically never end.

Lazy lists are created using thunks – the same thing you would use to mark a certain block of Ela code as a candidate for a lazy initialization.

Let's try to create an infinite list.

First we will need to declare a pretty typical `take` function that can take a specified amount of elements from a given list:

```
ela>let take 0 _ = []; n x::xs = x :: take (n - 1) xs; _ [] = []
```

Now we need to create our infinite list. We can do it through a function that accepts a number, increments it and uses the result to create list elements:

```
ela>let inf x = y :: (& inf y) where y = x + 1

ela>let lst = inf 0
```

OK, but what we can do with an infinite list? For example we can take a finite number of elements from it using our previously defined `take` function:

```
ela>take 10 lst
```

```
[1,2,3,4,5,6,7,8,9,10]
```

Or we can traverse this list until a certain condition is met:

```
ela>let traverse x::xs | x < 5 = x :: traverse xs | else = []; [] = []

ela>traverse (inf 0)

[1,2,3,4]
```

Another way to create lazy lists is with the help of concatenation operator `(++)`. Look at the following function:

```
ela>let cycle xs = xs ++ (& cycle xs)
```

I believe it is clear what this function does from its definition – it accepts a list and creates an infinite list based on it by cycling our initial list:

```
ela>cycle [1,2,3]

[1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,
1,2,3,1,2,3,1,2,3,1,2,3,1,2,3...]
```

However you should be careful with infinite lists. For example if your try to calculate the length of an infinite list using built-in `length` function your program will hung forever. That will actually happen with any function which works with lists in a strict manner. For example, a regular `fold` or `map` will fail on infinite lists too. That is the reason why standard Core module defines several "lazy" functions which are safe to use with infinite lists. These functions names normally end with a quote character. For example, a non-strict version of `map` is called `map'`, non-strict version of `filter` is called `filter'` and so forth.

```
ela>let inf2 = cycle [3,4,5]

ela>take 10 (map' (*2) inf2)

[6,8,10,6,8,10,6,8,10,6]
```

9.6. Ranges

Ranges in Ela are arithmetic sequences that can be used to generate lists. You only have to specify a first element, a last element, and (optionally) a second element that will be used to calculate stepping.

The syntax is pretty straightforward:

```
ela>[1..5]

[1,2,3,4,5]

ela>[1,3..10]

[1,3,5,7,9]

ela>[100,87..1]
```

```
[100,87,74,61,48,35,22,9]
```

An important note – if you want to create a range in a descending order than you have to specify the second element, e.g. the following code won't give you the expected result:

```
ela>[10..1]

[10]
```

This is how you need to change it to make it work:

```
ela>[10,9..1]

[10,9,8,7,6,5,4,3,2,1]
```

OK, I told you that in order to create a range you have to specify a first element, an optional second element and a last element? Well, that's not exactly accurate. The last element is optional as well. And you can generate infinite lists using ranges by simply omitting it:

```
ela>[1..]

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,2
8,29,30,31,32,33,34,35,36,37,38,39,40,41...]

ela>[3,2..]

[3,2,1,0,-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...]
```

Easy, isn't it?

> **Hint**
> Ela Console knows when to stop – if it sees that the value returned from your input is an infinite list it won't try to print out it as a whole (which is impossible because list is infinite). It will only pick first elements and print three dots to show that it displays only a part of it.

You can even create ranges with tuples:

```
ela>[(1,2)..(5,6)]

[(1,2),(2,3),(3,4),(4,5),(5,6)]
```

9.7. List comprehensions

Comprehension is a slightly more complex concept. Comprehension can be used to generate sequences of elements based on certain conditions. You can also perform transformation of selected values. Comprehensions are somewhat similar to C# Linq expressions but with a completely different notation.

Comprehension is combined from three parts – a value to select, an expression that actually fetches a value and an optional condition. Let's say that we have a range `[1..10]` and we need to select all even integers from it, multiply each element by two and compile a new list with the result. This is how this task might be written using Linq like syntax:

```
from x in [1..0] where x % 2 == 0 select x * 2
```

And that is the Ela version:

```
ela>[x * 2 \\ x <- [1..10] | x % 2 == 0]

[4,8,12,16,20]
```

What you can see here? First it starts from the selected expression (which is in our case `x * 2`), it is followed by a mandatory `(\\)` operator and an expression that denotes how we are going to obtain this x. In our case we are just simply fetching it from a numeric range. The comprehension ends with a guard that has a typical guard syntax – a boolean expression preceeded by pipe operator.

As I've mentioned above, guard is not mandatory and you can omit it:

```
ela>[x * 2 \\ x <- [1..10]]

[2,4,6,8,10,12,14,16,18,20]
```

You can also use multiple guards:

```
ela>[x * 2 \\ x <- [1..10] | x % 2 == 0, x > 5]

[12,16,20]
```

You can even generate your list based on several inputs like so:

```
ela>[x * y \\ x <- [10,9..1], y <- [1..10] | x % 2 == 0, x > 5]

[10,20,30,40,50,60,70,80,90,100,8,16,24,32,40,48,56,64,72,80,6,12,18,24,
30,36,42,48,54,60]
```

Comprehensions can also be used to generate lazy lists like so:

```
ela>[& x \\ x <- [1..]]

[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...]
```

to be continued...