Setup

- On **Windows** : Clone this Repository. You will find the command line program `BooleanCompiler.exe` in `KP-seminar-paper-Fsharp\BooleanCompiler\bin\Release\netcoreapp3.0\win-x64\publish`
- On **Linux**/**Mac** : Clone this Repository. You will need the dotnet core runtime. Then you can either execute the `Boolean Compiler.exe` in `KP-seminar-paper-Fsharp\BooleanCompiler\bin\Release\netcoreapp3.0` if the dotnet environment variable is set or you can execute the dll in the same folder with `dotnet BooleanCompiler.dll`

Usage

The `BooleanCompiler.exe` takes two command line arguments:

- `varBoolMap` is a Map on boolean values for the definition of the variables. It is of the form `'A:true|B:false|C:true'`.
- `stringtoParse` is the Boolean Expression, which should be parsed. It is of the form `'!A&(!!B|C)'`.

Example Call

Call: `./BooleanCompiler.exe 'A:true' 'A'`
Output: `Success: Var "A" evaluated as true`

Call: `./BooleanCompiler.exe 'A:true|B:false|C:true' '!A&(!!B|C)'`
Output: `Success: And (Not (Var "A"),Or (Not (Not (Var "B")),Var "C")) evaluated as false`

Comparison of functional F# to GO

Table of Contents

1. Introduction

2. F# Overview

F# is a multi paradigm programming language. That means it takes multiple parts of different code design and puts it together.
Primarily it was designed to be a functional programming language but you can also do pbject oriented programming or imperative programming
but the last one should only be rarely used and its considered to be bad design. F# was invented by a Microsoft Research Team and is compatible with
the whole .NET Plattform. So you can interact for example with Visual Basic or C#, which makes F# very powerfull for actual application writing.
This was the main idea, to build a functional programming language, that is integrated into the Microsoft Ecosystem.
Its language features are very similar to OCaml and it was indeed a role model for F#.
Because F# is a general purpose language it can be applied in a wide area. For example in web development, analytical programming and scripting.

1. Functional programming basics

The basic concept of functional programing is that the most important structural unit is a function in contrast for example to object oriented programming, where the
most important structural unit is a class. The abstraction happens through passing basic functions to higher level functions as arguments.
One goal of functional programming is also to minimze the use of mutable state. So in a functional programming language you only work with immutable data structures.

2. F# concepts

In this section we will focus on the propertys of F#'s language features. F# has a lot of language design features because it is a multi paradigm programming language. But in this section we will first and foremost focus on the functional programming features.

Let Bindings and immutability

```
let a = 10
a = 20
```

This code isn't valid F# code for two reasons. First if you bind a value to an identifier with the `let` binding you cannot
alter it anymore. It is immutable to assert one principle of functional programming. Second the `=` is the equality sign and not the assignment operator. This would be `<-`. You can do mutable data structures with `let mutable` but with this extra keyword its obviously discouraged.

Type interference

```
let a = 10
a = 20
```

Like we have seen above, we do not have to give a type definition for the variable `a`. F# derives it because the value `10` is of type `int` so `a` must be an `int` too. F# also can do this in more complex constructs like functions, consider this function definition.

```
let prefix prefixStr baseStr =
    prefixStr + "," + baseStr
```

F# interferes that the arguments `prefixStr` and `baseStr` have to be of type `string` and that the function returns a `string` because of the method body and the operation in which they are used. You can always add the type definition, if you want to. In this way F# combines the security of a strongly typed language and the read- and writability of a loosely typed language.

First-class functions

To be called a functional programming language, the programming languages functions have to fullfill the following propertys:

- Bind functions to identifiers
- Store functions in data structures like lists
- Pass functions as arguments in another function call
- Return a function from a function call

Of course F# supports all of the above propertys. The first one we have allready seen in the `let prefix` example. You can see an example of these propertys here.

The Pipe Operator

F# gives a convenient way to chain multiple functions together with the pipe operator `|>` and the forward composition operator `>>`. So you can easily build new functions from existing functions.

Example usage:

```
 let print message =
     printf "%s" message

"Hello World" |> print
```

Will print `"Hello World". This pipe operator takes the output of the left function and uses it as an argument for the right function. This is usefull when building more complex functions out of simpler ones.

Types

The type system in F# let us define data structures that have some sort of attributes. Together with functions you can encapsulate and polymorph your data without classes. For example the discriminate union type lets you easily define a class hierachy.

```
type Shape =
  // The value here is the radius.
| Circle of float
  // The value here is the side length.
| EquilateralTriangle of double
  // The value here is the side length.
| Square of double
  // The values here are the height and width.
| Rectangle of double * double
```

This shows another advantage of F#, which is it very short and readable code. If we had written this in language like Java, we needed much more code for the same structure.
To learn more about types you can look up this website.

3. Parser Example

The functional Boolean Parser in F# consists of three parts. The abstract syntax tree, the FParsec library, and the BooleanParser implementation.

1. Abstract syntax tree

The abstract syntax tree is very simple with the F# concepts. like we have seen in the previous section about Types, you can easily make class hierachys with discriminate unit types. This will result in the following.

```
type Node =
    | Or of Node * Node
    | And of Node * Node
    | Not of Node
    | Var of string
```

This means, that the type Node has four manifestations. It's either an `Or` or an `And`, which consists of a tuple of Nodes, a `Not`, which consists of one Node, or a `Var`, which is a string.
Now we have to add some functionality to evaluate a Node type. Because we know the manifestations of a Node we can use a typically functional programming paradigm, the pattern matching, which is done with the `match .. with` keyword in F#.

```
let rec eval(vars:Map<string,bool>) (ast:Node) : bool =
    match ast with
        | Or(lhs,rhs) -> eval vars lhs || eval vars rhs
        | And(lhs, rhs) -> eval vars lhs && eval vars rhs
        | Not(N) -> not(eval vars N)
        | Var(s) -> Map.find s vars
```

We define a recursive functions that gets a variable map to evaluate the Var Nodes and a root node to start with.

2. FParsec

To understand the Boolean Parser implementation we have first to understand the FParsec library and how it implements its parser combinators.

The Parser type

We define a Parser as a function of the form `string -> Result<'a>`. The string can be other kind of input but this is the basic definition, that let us combine different parsers. But what is the Result type?

The Result type

```
type Result<'a> =
    | Success of 'a
    | Failure of string
```

A Result is a discriminate union of Success of type 'a or Failure of type string. In the Fparsec library this result type has one mor generic type parameter, which defines the user state, but isn't important for understanding the functionality of the parser.

Summarized a Parser returns a Result of a type of your choosing embedded in Success or a Failure with an error message. In the Boolparser implementation we want to parse the string into a `Node` type, to build an AST and evaluating it afterwards.

Combinators

We can combine different parsers together with the parser combinators of FParsec. Some important ones are described here

- `ParserA .>>. ParserB`: executes ParserA and then Parser B and combines their `Result` in a tuple
- `ParserA >>. ParserB`: executes ParserA and then Parser B and returns the `Result` of `ParserB`. There is also a `.>>` combinator, to return the result of `ParserA`.
- `ParserA |>> func f`: applies the function `f` to the `Result Success` type. This is used to construct the AST.

These were some of the important combinators. To learn more about FParsec and parser combinators you can look up the FParsec website and this blog post about parser combinators.

3. Boolean Parser

In this section we will look upon the Boolean Parser implementation. We will focus on some of the basic ideas.The full code can be viewed in `BooleanCompiler/BooleanParser.fs`.

Combinating

We constuct new parsers with the help of the FParsec library like already seen in the last section. For example we define a `identifer` parser.

```
let parseIdentifier:Parser<string, unit> = ws >>. many1SatisfyL isAsciiLetter
"identifier"
```

It parses zero or more whitespace characters but doesn't return it in the Result and then parses at least one of the following characters, if they are ASCII characters.

Constructing the AST

To parse the input string into an AST Node we need the `|>>` Operator. For example:

```
let parseVariable = (parseIdentifier |>> fun x -> Var(x))
```

This parses a variable like we have seen above and then applys the lambad expression to the `Result` of the `parseidentifier`. so the signature of this Parser is `string -> Result<Node>` because we transform the `node` output of `parseidentifier` into a `Node`

Recursion

4. Comparison to Golang

5. Conclusion

6. References

- Smith, Chris (2009). "Programming F#". O'Reilly.
- Official website The F# Software Foundation
- Fsharpforfunandprofit
- FParsec website