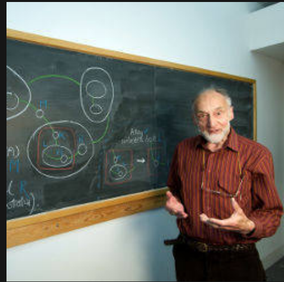
TABLE OF CONTENTS

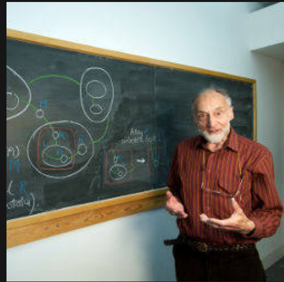
1 A little bit of history
2 F# and .NET
3 F# and .NET core
4 F# Vs < X >
5 In the Industry
6 To sum it up : standout features
7 Some code
8 Computation Expressions
9 *Active Patterns*
10 Web : "Full Stack"
11 Thanks for your attention
12 References

1 A LITTLE BIT OF HISTORY

(Syme, 2019)



Robin Milner, father of ML language

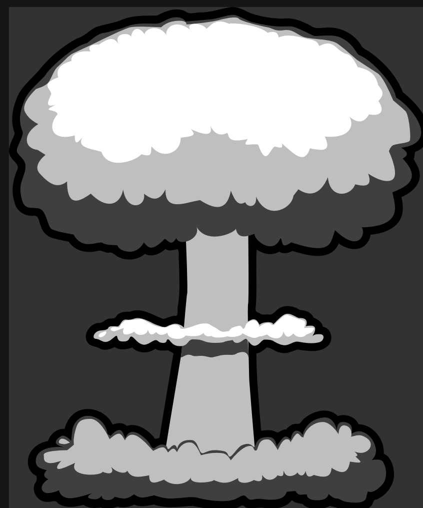


Robin Milner, father of ML language

Bill Gates, (very) successful businessman

When both worlds collide...

When both worlds collide...



Kidding aside,

Année	Recherche	"Corporate"
1973	ML, le Lisp « typé »	
1975		Microsoft
1995		Java
1997	Standard ML	
1998	Don Syme @ Microsoft Research	
2002		.NET (Kennedy and Syme, 2001)
2005	F#	

2 F# AND .NET

- F# is seen like THE functional language by MS on .NET
- .NET and F# co-evoluted

3 F# AND .NET CORE

- .NET Reboot
- 100% cross platform (Mac/Windows/Linux + x86/ARM)
- OSS

4 F# VS < X >

PROS

- *Stable* : no breaking changes in the language, good binary compatibility over the years
- *Coherent* : There is a straight and streamlined way to code in f#, hassle-free and no "zillions of ways to do the same thing", enforced by crystal clear guidelines, documentations and IDEs
- *Open* : biggest core dev are MS, but active, open contributions from the community, RFC proposals and so on...
- *fully interoperable* : thought and conceived *for* interoperability from the ground up. interop is F#'s DNA.

CONS

- *Functionnalities* (not on par with more pure FP focused languages like Haskell or Scala, no HKT)
- *No "killer app"* like Spark for Scala
- *Poor academic coverage*, especially in some countries where MS' reputation doesn't bode well with "research"

5 IN THE INDUSTRY

- MS, of course (Cloud/Azyre and internal products)
- Jet.com
- Genetec
- Bank/Trading
- Mobile dev (Xamarin)

6 TO SUM IT UP : STANDOUT FEATURES

- Type providers
- Computation expressions
- Active Patterns

7 SOME CODE

TYPE PROVIDERS

(Syme, Battocchi *et al.*, 2012)

Following slides courtesy of official MS doc

An F# type provider is a component that provides types, properties, and methods for use in your program. Type Providers generate what are known as Provided Types, which are generated by the F# compiler and are based on an external data source.

For example, an F# Type Provider for SQL can generate types representing tables and columns in a relational database. In fact, this is what the `SQLProvider` Type Provider does.

Provided Types depend on input parameters to a Type Provider. Such input can be a sample data source (such as a JSON schema file), a URL pointing directly to an external service, or a connection string to a data source.

A Type Provider can also ensure that groups of types are only expanded on demand; that is, they are expanded if the types are actually referenced by your program. This allows for the direct, on-demand integration of large-scale information spaces such as online data markets in a strongly typed way.

GENERATIVE TYPE PROVIDERS

Generative Type Providers produce types that can be written as .NET types into the assembly in which they are produced. This allows them to be consumed from code in other assemblies. This means that the typed representation of the data source must generally be one that is feasible to represent with .NET types.

ERASING TYPE PROVIDERS

Erasing Type Providers produce types that can only be consumed in the assembly or project they are generated from. The types are ephemeral; that is, they are not written into an assembly and cannot be consumed by code in other assemblies. They can contain delayed members, allowing you to use provided types from a potentially infinite information space. They are useful for using a small subset of a large and interconnected data source.

COMMONLY USED TYPE PROVIDERS

COMMONLY USED TYPE PROVIDERS

`FSharp.Data` includes Type Providers for JSON, XML, CSV, and HTML document formats and resources.

COMMONLY USED TYPE PROVIDERS

`FSharp.Data` includes Type Providers for JSON, XML, CSV, and HTML document formats and resources.

`SQLProvider` provides strongly-typed access to relation databases through object mapping and F# LINQ queries against these data sources.

TYPE PROVIDERS I : THE GOOD OLD CSV...

In [4]:

```
open FSharp.Data

type Gbpusd = CsvProvider<const(__SOURCE_DIRECTORY__ + "/data/gbpusd.csv")>
```

In [5]:

```
// Get the day where the rate lost more than 10% for the pound sterling
let gbpusd = Gbpusd.GetSample().Rows
gbpusd
|> Seq.pairwise
|> Seq.filter (fun (before, after) -> before.GbpUsd - after.GbpUsd > 0.1M)
|> Seq.map (fun (before,_) -> before.Date.ToShortDateString())
```

Out[5]:

```
seq ["06/23/2016"]
```

In [7]:

```
open Deedle

let titanic = Frame.ReadCsv(__SOURCE_DIRECTORY__ + "/data/titanic.csv")
```

In [51]:

```
// Get the survival rate by class
let byClass =
    titanic
     |> Frame.groupRowsByString "Pclass"
     |> Frame.getCol "Survived" :> Series<(string * int),bool>
     |> Series.applyLevel fst (Series.values >> (Seq.countBy id) >> series)
     |> Frame.ofRows
     |> Frame.sortRowsByKey
     |> Frame.indexColsWith [ "Died"; "Survived" ]
```

In [9]:

```
byClass?Total <- byClass?Died + byClass?Survived
```

In [51]:

```
// Get the survival rate by class
let byClass =
    titanic
    |> Frame.groupRowsByString "Pclass"
    |> Frame.getCol "Survived" :> Series<(string * int),bool>
    |> Series.applyLevel fst (Series.values >> (Seq.countBy id) >> series)
    |> Frame.ofRows
    |> Frame.sortRowsByKey
    |> Frame.indexColsWith [ "Died"; "Survived" ]
```

In [9]:

```
byClass?Total <- byClass?Died + byClass?Survived
```

In [10]:

```
frame [ "Died (%)" => round (byClass?Died / byClass?Total * 100.0)
        "Survived (%)" => round (byClass?Survived / byClass?Total * 100.0) ]
```

Out[10]:

	Died (%)	Survived (%)
1	37	63
2	53	47

TYPE PROVIDER II : OPEN DATA

In [11]:

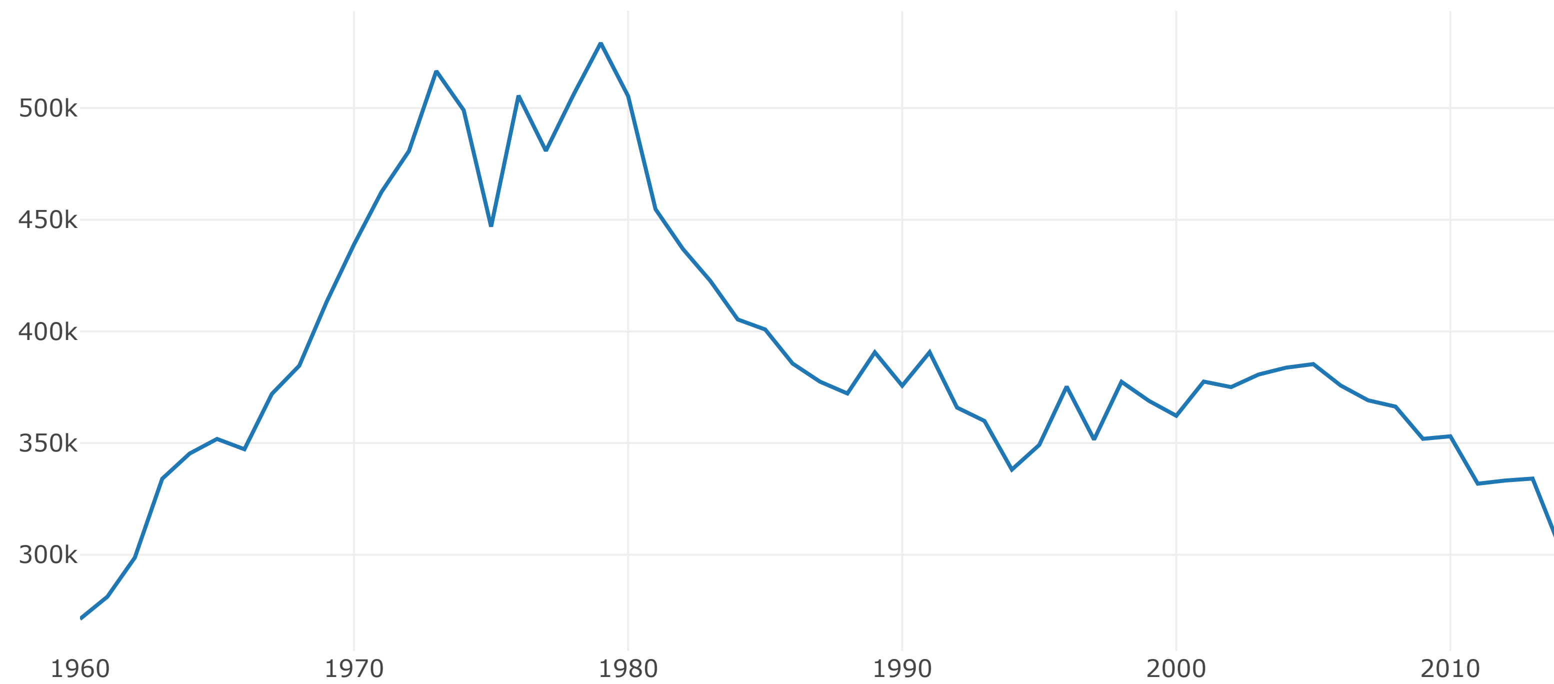
```
open XPlot.Plotly

let wb = WorldBankData.GetDataContext()
wb
```

Out[11]: FSharp.Data.Runtime.WorldBank.WorldBankData

In [12]:

```
// Get the yearly CO2 emissions for France, Germany and USA
wb.Countries.France.Indicators.``CO2 emissions (kt)``
|> Chart.Line
```

Out[12]:



TYPE PROVIDERS III : JSON

```
"http://api.openweathermap.org/data/2.5/forecast/d
q=Prague&mode=json&units=metric&cnt=10&APPID=cb63a
```


```
{
  "city": {
    "id": 3067696,
    "name": "Prague",
    "coord": {
      "lon": 14.4213,
      "lat": 50.0875
    },
[...]
```

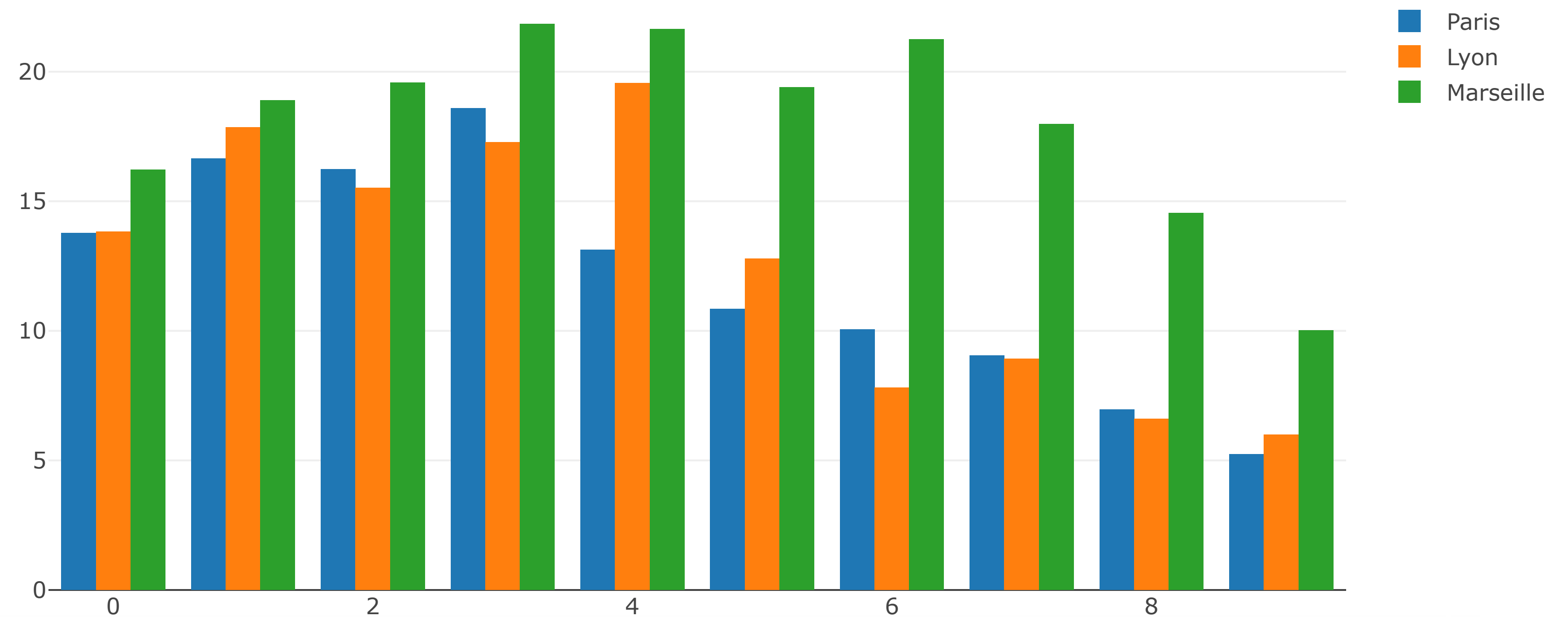
In [13]:

```
type W = JsonProvider<"http://api.openweathermap.org/data/2.5/forecast/daily?q=

let getTemps city =
  let w = W.Load("http://api.openweathermap.org/data/2.5/forecast/daily?q=" + c
  [ for d in w.List -> d.Temp.Day ]

let cities = ["Paris";"Lyon";"Marseille"]
```

In [14]:

```
// Display a temperature barplot for cities
cities
|> Seq.map (getTemps >> Seq.indexed)
|> Chart.Column
|> Chart.WithLabels cities
```

Out[14]:



TYPE PROVIDERS III : SQL

In [15]:

```
[<Literal>]
let ConnectionString =
    "Data Source=" +
    __SOURCE_DIRECTORY__ + @"/data/northwindEF.db;" +
    "Version=3;foreign keys=true"

[<Literal>]
let ResolutionPath = __SOURCE_DIRECTORY__ + @"/local"

open FSharp.Data.Sql

type Sql = SqlDataProvider<
                Common.DatabaseProviderTypes.SQLITE,
                SQLiteLibrary = Common.SQLiteLibrary.SystemDataSQLite,
                ConnectionString = ConnectionString,
                ResolutionPath = ResolutionPath,
                CaseSensitivityChange = Common.CaseSensitivityChange.ORIGINAL>

let db = Sql.GetDataContext()
```

In [16]:

```
// clients name list
db.Main.Customers
|> Seq.map (fun c -> c.ContactName)
|> Seq.toList
```

Out[16]:

```
["Maria Anders"; "Ana Trujillo"; "Antonio Moreno"; "Thomas Hardy";
 "Christina Berglund"; "Hanna Moos"; "Frédérique Citeaux"; "Martín Somme
r";
 "Laurence Lebihan"; "Elizabeth Lincoln"; "Victoria Ashworth";
 "Patricio Simpson"; "Francisco Chang"; "Yang Wang"; "Pedro Afonso";
 "Elizabeth Brown"; "Sven Ottlieb"; "Janine Labrune"; "Ann Devon";
 "Roland Mendel"; "Aria Cruz"; "Diego Roel"; "Martine Rancé"; "Maria Lars
son";
 "Peter Franken"; "Carine Schmitt"; "Paolo Accorti"; "Lino Rodriguez";
 "Eduardo Saavedra"; "José Pedro Freyre"; "André Fonseca"; "Howard Snyde
r";
 "Manuel Pereira"; "Mario Pontes"; "Carlos Hernández"; "Yoshi Latimer";
 "Patricia McKenna"; "Helen Bennett"; "Philip Cramer"; "Daniel Tonini";
 "Annette Roulet"; "Yoshi Tannamuri"; "John Steel"; "Renate Messner";
 "Jaime Yorres"; "Carlos González"; "Felipe Izquierdo"; "Fran Wilson";
 "Giovanni Rovelli"; "Catherine Dewey"; "Jean Fresnière"; "Alexander Feue
r";
 "Simon Crowther"; "Yvonne Moncada"; "Rene Phillips"; "Henriette Pfalzhei
m";
 "Marie Bertrand"; "Guillermo Fernández"; "Georg Pipps"; "Isabel de Castr
```

TYPE PROVIDER IV : R (!)

In [18]:

```
#I "./local/RProvider/bin"
#r "DynamicInterop.dll"
#r "RDotNet.dll"
#r "RDotNet.FSharp.dll"
#r "RProvider.dll"
#r "RProvider.Runtime.dll"
open RDotNet
open RProvider
open RProvider.``base``
open RProvider.stats
open RProvider.graphics
open RProvider.svglite
open RProvider.grDevices
```

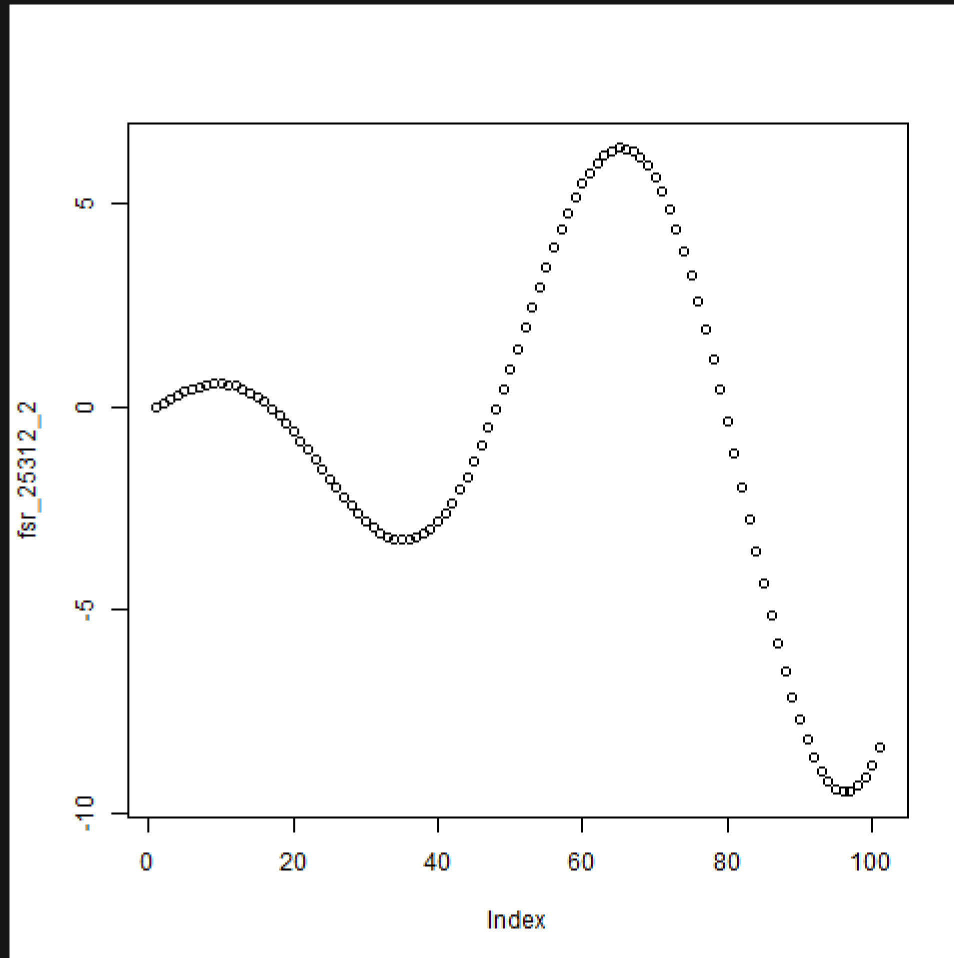
In [19]:

```
// little utility function to display R plot in the notebook
open System.IO
let plotR (f : Lazy<SymbolicExpression>) = // f est une fonction de plot non év
    let path = Path.GetTempFileName()
    R.png(path) |> ignore
    f.Force() |> ignore
    R.dev_off() |> ignore
    let output = Util.Image path
    File.Delete(path)
    output
```

In [20]:

```
let data = [ for x in 0. .. 0.1 .. 10. -> x * cos x ]
lazy (R.plot data) |> plotR
```

Out[20]:

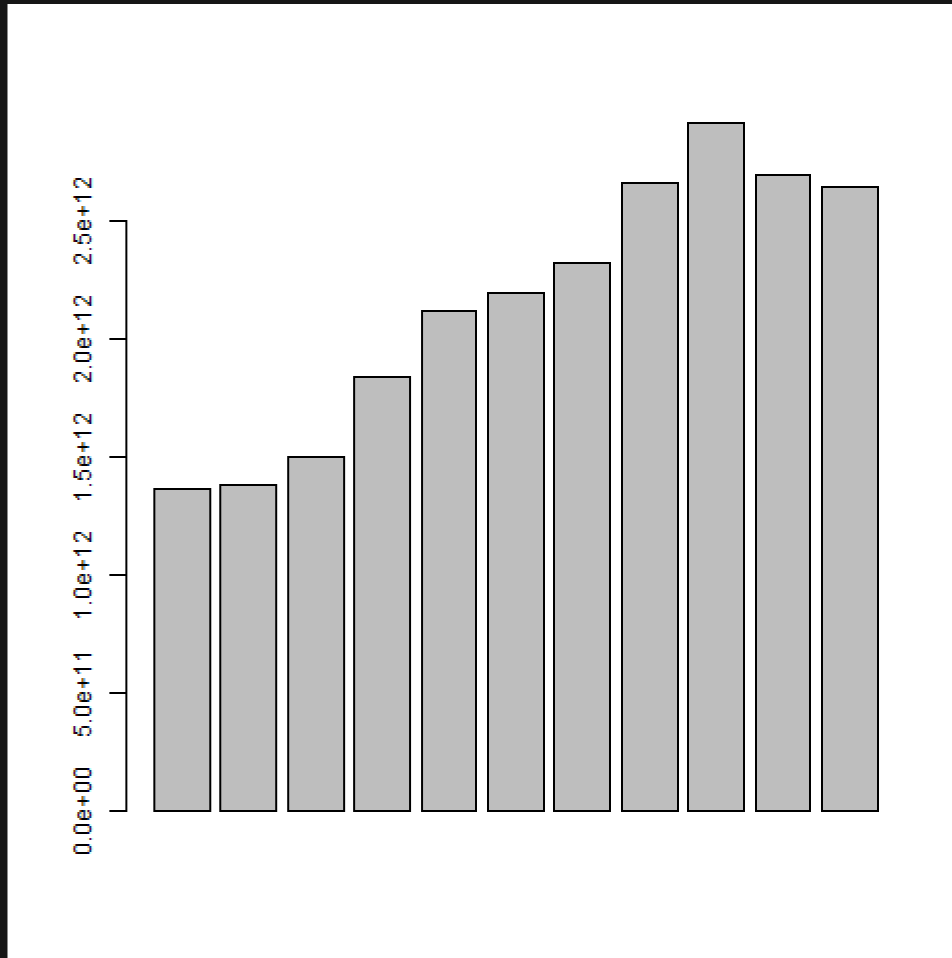


Microsoft R Open 3.5.2

In [21]:

```
lazy ([ for y in 2000 .. 2010 -> wb.Countries.France.Indicators.``GDP (current
       |> R.barplot)
|> plotR
```

Out[21]:



In [22]:

```
let sample = query {
    for c in wb.Countries do
    where (c.Indicators.``Population, total``.[2010] > 100000000.)
    select c.Indicators } |> Seq.toArray
```

In [22]:

```
let sample = query {
    for c in wb.Countries do
    where (c.Indicators.``Population, total``.[2010] > 100000000.)
    select c.Indicators } |> Seq.toArray
```

In [23]:

```
let growth = [ for c in sample -> c.``GDP growth (annual %)``.[2010]]
let gender = [ for c in sample -> c.``Employment to population ratio, 15+, fema
let youth = [ for c in sample -> c.``Population ages 0-14 (% of total populatio
let pollution = [ for c in sample -> c.``CO2 emissions (kg per PPP $ of GDP)``.
```

In [24]:

```
// Transform those 4 columns into a dataframe and display in R
lazy(namedParams [
        "Growth", growth
        "Gender", gender
        "Youth", youth
        "Pollution", pollution ]
    |> R.data_frame
    |> R.plot)
|> plotR
```

Out[24]:

8 COMPUTATION EXPRESSIONS

Monadically yours… (Petricek and Syme, 2014)

> *Computation expressions in F# provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. Depending on the kind of computation expression, they can be thought of as a way to express monads, monoids, monad transformers, and applicative functors.*

> *However, unlike other languages (such as do-notation in Haskell), they are not tied to a single abstraction, and do not rely on macros or other forms of metaprogramming to accomplish a convenient and context-sensitive syntax.*

Source : https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions

COMPUTATION EXPRESSIONS I : FROM PIPES TO `Seq`

COMPUTATION EXPRESSIONS I : FROM PIPES TO `Seq`

In [25]:

```
// Get the infinity sequence of natural numbers
let naturals =
    let rec nat k =
        seq {
            yield k
            yield! nat (k + 1)
        }
    nat 0
```

COMPUTATION EXPRESSIONS I : FROM PIPES TO `Seq`

In [25]:

```
// Get the infinity sequence of natural numbers
let naturals =
    let rec nat k =
        seq {
            yield k
            yield! nat (k + 1)
        }
    nat 0
```

In [27]:

```
naturals
|> Seq.map (fun x -> x * x)
|> Seq.take 10
|> Seq.iter (printfn "%d")
```

```
0
1
```

COMPUTATION EXPRESSIONS II : LINQ OR THE SQL "MONAD"

In [28]:

```
open System.Linq
type ComptePays = { Pays : string; Nombre : int }

// Return the top five countries with the most clients, ordered by clients numb
query {
    for c in db.Main.Customers do
    groupBy c.Country into y
    sortByDescending (y.Count())
    take 5
    select { Pays = y.Key ; Nombre = y.Count() }
} |> Util.Table
```

Out[28]:

Pays	Nombre
USA	13
France	11
Germany	11
Brazil	9
UK	7

COMPUTATION EXPRESSIONS III : ASYNC

In [30]:

```
open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]
```

In [31]:

```
let fetchAsync(name, url:string) =
    async {
        try
            let uri = System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }
```

In [32]:

```
// do a runAll() function which launchs all of those urls asynchronously
let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()
```

```
Read 117495 characters for Bing
Read 160799 characters for Microsoft.com
Read 39690 characters for MSDN
```

CUSTOM COMPUTATION EXPRESSIONS I : MAYBE MONAD

In [33]:

```
type Maybe() =
    member __.Bind(p, rest) =
        match p with
        | None -> None
        | Some a -> rest a

    member __.Return(p) =
        Some p

let maybe = Maybe()
```

In [34]:

```
let tryDecr x n =
  printfn "Conditionally decrementing %A by %A" x n
  if x > n then Some (x - n) else None
tryDecr
```

Out[34]: `<fun:it@4-13> : (int -> (int -> FSharpOption`1))`

In [35]:

```
let maybeDecr x = maybe {
    let! y = tryDecr x 10
    let! z = tryDecr y 30
    let! t = tryDecr z 50
    return t
  }
maybeDecr
```

Out[35]:

```
<fun:it@7-14> : (int -> FSharpOption`1)
```

In [36]:

```
maybeDecr 100 |> printfn "%A"
maybeDecr 50  |> printfn "%A"
maybeDecr 30  |> printfn "%A"
```

```
Conditionally decrementing 100 by 10
Conditionally decrementing 90 by 30
Conditionally decrementing 60 by 50
Some 10
Conditionally decrementing 50 by 10
Conditionally decrementing 40 by 30
Conditionally decrementing 10 by 50
<null>
Conditionally decrementing 30 by 10
Conditionally decrementing 20 by 30
<null>
```

CUSTOM COMPUTATION EXPRESSIONS II : `Eventually`

- Encapsulates a computation as a series of steps that can be evaluated one step at a time.
- Encodes the error state of the expression as evaluated so far with a discriminated union type, OkOrException

In [37]:

```
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)

module Eventually =
    // The bind for the computations. Append 'func' to the
    // computation.
    let rec bind func expr =
        match expr with
        | Done value -> func value
        | NotYetDone work -> NotYetDone (fun () -> bind func (work()))

    // Return the final value wrapped in the Eventually type.
    let result value = Done value

    type OkOrException<'T> =
        | Ok of 'T
        | Exception of System.Exception

    // The catch for the computations. Stitch try/with throughout
    // the computation, and return the overall result as an OkOrException.
    let rec catch expr =
        match expr with
        | Done value -> result (Ok value)
```

In [38]:

```
// Loop from 1 to 2 and display " x = i " (for i = 1,2) and then return additio
// Try 1,2 and 4 steps

let step x = Eventually.step x
```

In [39]:

```
let comp = eventually {
    for x in 1..2 do
        printfn " x = %d" x
    return 3 + 4 }
```

In [39]:

```
let comp = eventually {
    for x in 1..2 do
        printfn " x = %d" x
    return 3 + 4 }
```

In [40]:

```
// returns "NotYetDone <closure>"
comp |> step
```

Out[40]:

```
NotYetDone <fun:bind@11>
```

In [39]:

```
let comp = eventually {
    for x in 1..2 do
        printfn " x = %d" x
    return 3 + 4 }
```

In [40]:

```
// returns "NotYetDone <closure>"
comp |> step
```

Out[40]: NotYetDone <fun:bind@11>

In [41]:

```
// prints "x = 1"
// returns "NotYetDone <closure>"
comp |> step |> step
```

Out[41]: NotYetDone <fun:bind@11>

```
 x = 1
```

In [42]:

```
// prints "x = 1"
// prints "x = 2"
// returns "Done 7"
comp |> step |> step |> step |> step
```

Out[42]: Done 7

```
x = 1
x = 2
```

9 *ACTIVE PATTERNS*

(Syme, Neverov *et al.*, 2007)

> *Active patterns enable you to define named partitions that subdivide input data, so that you can use these names in a pattern matching expression just as you would for a discriminated union. You can use active patterns to decompose data in a customized manner for each partition.*

Source : https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/active-patterns

ACTIVE PATTERNS I : "NORMAL" CASE

In [43]:

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd

let TestNumber input =
   match input with
   | Even -> printfn "%d is even" input
   | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

```
7 is odd
11 is odd
32 is even
```

In [44]:

```
open System.Drawing

let (|RGB|) (col : Color) =
     ( col.R, col.G, col.B )

let (|HSB|) (col : Color) =
   ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let printRGB (col: Color) =
   match col with
   | RGB(r, g, b) -> printfn " Red: %d Green: %d Blue: %d" r g b

let printHSB (col: Color) =
   match col with
   | HSB(h, s, b) -> printfn " Hue: %f Saturation: %f Brightness: %f" h s b

let printAll col colorString =
  printfn "%s" colorString
  printRGB col
  printHSB col
```

In [45]:

```
printAll Color.Red "Red"
printAll Color.Black "Black"
printAll Color.White "White"
printAll Color.Gray "Gray"
printAll Color.BlanchedAlmond "BlanchedAlmond"
```

```
Red
 Red: 255 Green: 0 Blue: 0
 Hue: 0.000000 Saturation: 1.000000 Brightness: 0.500000
Black
 Red: 0 Green: 0 Blue: 0
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.000000
White
 Red: 255 Green: 255 Blue: 255
 Hue: 0.000000 Saturation: 0.000000 Brightness: 1.000000
Gray
 Red: 128 Green: 128 Blue: 128
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.501961
BlanchedAlmond
 Red: 255 Green: 235 Blue: 205
 Hue: 36.000000 Saturation: 1.000000 Brightness: 0.901961
```

ACTIVE PATTERNS II : PARTIAL

In [46]:

```
let (|Integer|_|) (str: string) =
   let mutable intvalue = 0
   if System.Int32.TryParse(str, &intvalue) then Some(intvalue)
   else None

let (|Float|_|) (str: string) =
   let mutable floatvalue = 0.0
   if System.Double.TryParse(str, &floatvalue) then Some(floatvalue)
   else None

let parseNumeric str =
   match str with
     | Integer i -> printfn "%d : Integer" i
     | Float f -> printfn "%f : Floating point" f
     | _ -> printfn "%s : Not matched." str
```

In [47]:

```
parseNumeric "1.1"
parseNumeric "0"
parseNumeric "0.0"
parseNumeric "10"
parseNumeric "Something else"
```

```
1.100000 : Floating point
0 : Integer
0.000000 : Floating point
10 : Integer
Something else : Not matched.
```

Sometimes, you need to partition only part of the input space. In that case, you write a set of partial patterns each of which match some inputs but fail to match other inputs. Active patterns that do not always produce a value are called partial active patterns; they have a return value that is an option type. To define a partial active pattern, you use a wildcard character (_) at the end of the list of patterns inside the banana clips. The following code illustrates the use of a partial active pattern.

ACTIVE PATTERNS III : PARAMETRIZED

In [48]:

```
open System.Text.RegularExpressions

// ParseRegex parses a regular expression and returns a list of the strings tha
// the regular expression.
// List.tail is called to eliminate the first element in the list, which is the
// since only the matches for each group are wanted.
let (|ParseRegex|_|) regex str =
    let m = Regex(regex).Match(str)
    if m.Success
    then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None
```

Active patterns always take at least one argument for the item being matched, but they may take additional arguments as well, in which case the name parameterized active pattern applies.

Active patterns always take at least one argument for the item being matched, but they may take additional arguments as well, in which case the name parameterized active pattern applies.

Additional arguments allow a general pattern to be specialized. For example, active patterns that use regular expressions to parse strings often include the regular expression as an extra parameter, as in the following code, which also uses the partial active pattern Integer defined in the previous code example.

In this example, strings that use regular expressions for various date formats are given to customize the general ParseRegex active pattern. The Integer active pattern is used to convert the matched strings into integers that can be passed to the DateTime constructor.

In [49]:

```
// Three different date formats are demonstrated here. The first matches two-
// digit dates and the second matches full dates. This code assumes that if a t
// date is provided, it is an abbreviation, not a year in the first century.
let parseDate str =
    match str with
      | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{1,2})$" [Integer m; Integer d; Integ
          -> new System.DateTime(y + 2000, m, d)
      | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{3,4})" [Integer m; Integer d; Intege
          -> new System.DateTime(y, m, d)
      | ParseRegex "(\d{1,4})-(\d{1,2})-(\d{1,2})" [Integer y; Integer m; Intege
          -> new System.DateTime(y, m, d)
      | _ -> new System.DateTime()
```

In [50]:

```
let dt1 = parseDate "12/22/08"
let dt2 = parseDate "1/1/2009"
let dt3 = parseDate "2008-1-15"
let dt4 = parseDate "1995-12-28"

printfn "%s %s %s %s" (dt1.ToString()) (dt2.ToString()) (dt3.ToString()) (dt4.To
```

```
12/22/2008 00:00:00 01/01/2009 00:00:00 01/15/2008 00:00:00 12/28/1995 0
0:00:00
```

10 WEB : "FULL STACK"

- SAFE Stack
- WebSharper

FABLE : STATE OF THE ART JS TRANSPILER

(Nunez and Fahad, 2016)

FABLE : STATE OF THE ART JS TRANSPILER

(Nunez and Fahad, 2016)

- With all the goodies from FP + static typing

FABLE : STATE OF THE ART JS TRANSPILER

(Nunez and Fahad, 2016)

- With all the goodies from FP + static typing
- Clean JS output, with recent standards (ES2015+)

FABLE : STATE OF THE ART JS TRANSPILER

(Nunez and Fahad, 2016)

- With all the goodies from FP + static typing
- Clean JS output, with recent standards (ES2015+)
- Easy interop with JS ecosystem (npm/yarn)

FABLE : STATE OF THE ART JS TRANSPILER

(Nunez and Fahad, 2016)

- With all the goodies from FP + static typing
- Clean JS output, with recent standards (ES2015+)
- Easy interop with JS ecosystem (npm/yarn)
- Almost all FSharp "core", efficient pruning on the source code

SAFE-STACK

SAFE-STACK

- Almost "isomorphic" Client-Server programming model

SAFE-STACK

- Almost "isomorphic" Client-Server programming model
- Based on Model/View/Update from `elm`

SAFE-STACK

- Almost "isomorphic" Client-Server programming model

- Based on Model/View/Update from `elm`

- Model — the state of your application
- Update — a way to update your state
- View — a way to view your state as HTML

- FRP with `React` (`HMR` support and so on.)

Hot Module Replacement (HMR) allows to update the UI of an application while it is running, without a full reload. In SAFE stack apps, this can dramatically speed up the development for web and mobile GUIs, since there is no need to "stop" and "reload" and application. Instead, you can make changes to your views and have them immediately update in the browser, without the need to restart the application.

11 THANKS FOR YOUR ATTENTION

12 REFERENCES

(Syme, 2019) Syme Don, ``_The Early History of F# (HOPL IV-second draft)_'', , vol. , number , pp. , 2019. online

(Kennedy and Syme, 2001) A. Kennedy and D. Syme, ``_Design and implementation of generics for the. net common language runtime_'', ACM SigPlan Notices, 2001.

(Syme, Battocchi *et al.*, 2012) Syme Don, Battocchi Keith, Takeda Kenji *et al.*, ``_Strongly-typed language support for internet-scale information sources_'', Technical Report MSR-TR-2012--101, Microsoft Research, vol. , number , pp. , 2012.

(Petricek and Syme, 2014) T. Petricek and D. Syme, ``_The F# computation expression zoo_'', International Symposium on Practical Aspects of Declarative Languages, 2014.

(Syme, Neverov *et al.*, 2007) D. Syme, G. Neverov and J. Margetson, ``_Extensible pattern matching via a lightweight language extension_'', ACM SIGPLAN Notices, 2007.

(Nunez and Fahad, 2016) Alfonso Garcia-Caro Nunez and Suhaib Fahad, ``_Mastering F#_'', 2016.