Some F# Practicalities

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

How to Develop F# Programs

F# programs are just text files

You can create and edit them with the text editor of your choice

F# files should end with "`.fs`"

You can either

Batch compile into a "`.exe`" file with `fsc`, and run:

```
>fsc file.fs
>file.exe
```

Or, use the the F# interactive compiler, `fsi`

The F# Interactive Compiler

```
>fsi

F# Interactive for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License

For help type #help;;

>
```

Gives you an environment where you can type F# expressions to the prompt, and have them evaluated. End every expression with "`;;`"

```
> 5 + 6 ;;
val it : int = 11
>
```

So `fsi` can be used as a simple calculator (read - eval - print)

The F# Interactive Compiler (2)

Any F# expression can be evaluated:

```
> let x = 17.0 in x*(3.0 + 7.0/x);;
val it : float = 58.0
```

You can also make declarations with `let`. These are visible from then on:

```
> let x = 17.0;;
val x : float = 17.0
> x + 33.5;;
val it : float = 50.5
```

The F# Interactive Compiler (3)

You will want to use `fsi` for interactive testing. To get your code into `fsi`, use the `#load` command:

```
#load "file.fs";;
```

This will compile the code in `file.fs` and load it into `fsi`

`fsi` will create a *module* named `File`, where the declared entities in `file.fs` reside (more on modules later)

The F# Interactive Compiler (4)

A function `f`, declared in `file.fs`, can be accessed by prefixing its name with the module name:

```
> File.f 2;;
val it : int = 47
```

To avoid the prefix, you can first *open* the module:

```
> open File;;
> f 2;;
val it : int = 47
```

Visual Studio

Windows users can use Visual Studio

From Visual Studio 2010 there is full support for F#

`fsi` can also be run from Visual Studio

Testing with FsCheck

A tool to do **property-based testing** of .NET programs

A .NET version of `Quickcheck`, originally created for the functional language Haskell

You will use it to check your solutions for Lab 1 and 2

(And, of course, you're welcome to use for Lab 3/4 and project as well)

FsCheck – How it Works

A function `Check.Quick` that takes a *property* as argument

The property is encoded as a function, say `p`, which returns a boolean (`true`/`false`)

(`p` is called a *predicate*)

The call `Check.Quick p` will then run `p` with random arguments

If there is an argument `x` such that `p x = false`, then the test *fails* for `x`

`Check.Quick` will then try to find a *smaller* argument where the test fails, and report that

Checking Against a Reference Implementation

Say we have written a function `f`

Assume that we already have a *reference implementation* `F`

How to check whether `f` always returns the same value as `F` for the same input:

```
let p x = f x = F x

Check.Quick p
```

Checking other Properties

Any property that can be encoded as a predicate can be checked

Example: checking whether the `length` function on lists always is non-negative:

```
let p l = length l >= 0

Check.Quick p
```

Using FsCheck to Test your Lab Assignments

A zip archive `labs.zip` to be downloaded from the course home page

Contains:

- `FsCheck`
- script `fsi.fsx` to load proper source files
- `Test.fs` with predicates to test your solutions against reference implementations
- `Program.fs` with a function `main` that tests all solutions at once
- Lab skeletons `Lab1.fs`, `Lab2.fs` that can be used as templates for your solutions

How to Use

Download the zip archive, and unpack where your project is

For Lab 1 load `Lab1.fs`, `Test.fs`, `Program.fs`

Run all tests using `Test.all` in `Test.fs` (or by compiling and executing the project)

Or: use the individual test functions in `Test.fs` one by one

Similarly for Lab 2

Advice: use the lab skeletons for your solutions. Then the names of your functions will fit the test predicates