FABLE Project Demos

FABLE is a new (still under development) F# to Javascript compiler. The documentation for FABLE is reasonably good, although the skeleton projects should teach you what is needed to do the project work. These two demos introduce FABLE and the development tools you will use in your project. Each introduces different technology, and `fable-electron-init` builds on `fable-hlp-start`.

You could use either of these as a template for your project work. Following this walkthrough will show you how they work and how to modify them.

- fable-hlp-start
- fable-electron-init

TINY-projects

This demo takes the code from Worksheet 4 and enhances it as follows:

- The tokeniser and parser implement a complete functional language TINY
- the AST is processed by a *bracket abstraction algorithm*, as in the lectures, to make a combinator representation of the language expressions
- The evaluator does *combinator reduction* - as in the lectures

where to find?

The code is not much larger than for Worksheet 4 (about 600 lines). It is extraordinary that a complete functional language compiler and run-time system can be written so compactly. That is partly due to the simplicity of pure functional languages (everything is a function) and partly because they are very expressive and able to implement code compactly.

The **same** code is used in 3 independent projects:

- VS Code: `tiny-vscode`
- Visual studio: `tiny-vs`
- FABLE: `tiny-fable-start`

Most of your project code will be written **and tested** under VS Code or Visual Studio (you choose) like this. It will then be run (to check it works, with less extensive testing) under FABLE as here. You will therefore end up using `tiny-fable-start` and one of the other two projects.

The tiny-fable-start project shows:

- How to make an F# project that compiles to Javascript using FABLE.
- How to use Javascript modules under **node.js** with the **yarn** pakag manager.
- How to run Javascript/HTML from a browser.
- How to use developer tools to inspect and debug running Javascript.
- How to configure FABLE to automatically recompile whenever source code changes.

Follow the `README.md` file in this demo for more information.

The two non-FABLE projects illustrate how to set up F# projects from source code. Working projects are included together with instructions for how to reconstruct the entire project from an empty directory and the source code.

The non-FABLE projects also show how to use expecto test framework which integrates FSUnit and FSCheck conveniently.

Testing

The tiny-vscode and tiny-vs projects include dependencies on expecto and some sample tests. Expecto is a *smooth test framework that is easy to use* that includes both FsCheck and FsUnit. For more information read this introduction or the expecto documentation.

Fable Electron Init

This is a skeleton project (very little code) that would make a starting point for your GUI and shows:

- How to use the CodeMirror Javascript editor component in an F# program.
- How to write a syntax coloring tokeniser for CodeMirror.
- How to use Electron to make a desktop application from HTML and F#.
- How to access desktop files etc from F# running under Electron
- How to automatically reload the app when files change.

Follow the README.md file in this demo for more information.

Project Workflow

Most of the project code can be written as pure F# and tested under VS Code or Visual studio as normal. You will write expecto tests to validate the source code.

Project GUI workflow

The combination of `yarn watch` and `gulp start` (run in separate terminal windows) will automatically recompile and then reload an F# application whenever any source file is changed. This makes development very smooth and arguably better than Visual Studio. Source files can be edited using VS Code. In fact Visual Studio could also be used, as an editor, with the same auto-compile and auto-run system via FABLE *in parallel* with F# compiling directly and running under a terminal within Visual Studio.

Caveats:

- It is easy to end up with orphaned taks running continuously that use CPU and make the system sluggish. Watch for these and kill them with task manager (under windows).
- The auto-reload is very nice when making small changes to code, or changing layout etc. Code that does not compile can prevent live reload working, as can deleting the client window. In this case just stop gulp (Ctrl-C Y) and restart it.

Project Segmentation

The `difficult` part of the project (in that it requires extensive testing) can be written as F# functions that run from a command-line and are developed separately from the GUI. Since this F# code will also compile with FABLE you have a choice of development environments. It would be wise to test using both F# compiler and FABLE, since FABLE is not precisely identical to the normal F# compiler and using it any problems can be caught early. If you write your code correctly you can have modules with testable code that can be loaded and tested under either FABLE or F# compiler from a (different) top-level file.

The GUI must be written using F# compiled with FABLE and run under electron, as in `tiny-fable-start`. Very little is needed (a few buttons perhaps) different from that provided here. The very large `Node.js` ecosystem provides you with resources for enhancement, also many (though not all) client-side web frameworks will be compatible. The project does not require any such complexity, but it provides one way to enhance the work.

Challenge

Write a TINY parser and bracket abstraction algorithm in the TINY language, capable of compiling itself. You may use the existing tokeniser and reduction engine (which in any case could not efficiently be written in a pure functional language). You make any changes you like to the F# code here to enhance the specification of TINY. This is an entirely possible task, but probably not doable in the time limits of Spring Term! 10 challenge points to anyone doing this whose individual project mark is A or higher (this prevents anyone sacrificing project work to chase these difficult marks). Up to 5 marks available for partial success here, e.g. just the bracket abstraction, which is much less work. Work must be unique (multiple similar copies will all get no marks).