Homework 8

Due Sunday, May 10 by 11:59pm

Turn-In Instructions

For the specification portion of this assignment, provide a LaTeX source file and pre-built PDF. For full credit, your LaTeX file should build properly. Be sure to `git add` all necessary files (e.g., images) if your LaTeX depends on it. Your specification should be stored in the `spec` folder.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw_project_<your user name>`. For example, my repository would be `cs334hw_project_dbarowy`. You should have received an invite to commit in the repository in your email. If you did not receive this email, please contact me right away!

A good way to check that you submitted your assignment correctly is to `git clone` your repository to a new folder and then try building/running everything.

Pair Programming and Honor Code: You may optionally collaborate with one other person on the submission for this assignment. If you work with a partner, choose one of the two partner repositories for your submission. In the other repository, be sure to leave a `collaborators.txt` file that states who you worked with and the name of the repository where the code may be found. If you would like me to pair you with a partner, please let me know. You may not share code with any student who is not your designated partner.

Upon Completion: Please update the `README.md` file to let me know that you are finished.

This assignment is due on Sunday, May 10 by 11:59pm.

Problems

Q1. (*70 points*) ..... Minimal Project Prototype

For this assignment, you will build a minimally working version of your language. Along the way, you should also update your project specification.

A minimally working interpreter has the following components:

(a) A parser. Put your parser in a library file called `ProjectParser.fs`. The namespace for the parser should also be called `ProjectParser`.

(b) An interpreter / evaluator. Put your interpreter in a library file called `ProjectInterpreter.fs`. The namespace for the interpreter should also be called `ProjectInterpreter`.

(c) A driver program. The driver should contain a `main` method that takes input from the user, parses and interprets it using the appropriate library calls, and displays the result. Put your `main` method in a file called `Program.fs`. For example, if your project is an infix scientific calculator (an expression-oriented language), it might accept input and return a result on the command line as follows:

```
$ dotnet run "1 + 2"
3
```

Alternatively, your language might read in a text file that contains the same program as above, e.g.,

```
$ dotnet run myfile.calc
3
```

Either way, running your language without any input should produce a helpful "usage" message that explains how to use your programming language.

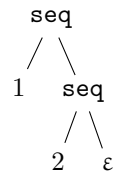
```
$ dotnet run
Usage:
  dotnet run <file.calc>

  Calculang will frobulate your foobars.
```

You should think carefully about what constitutes a "primitive value" in your language. Primitive values and operations on primitive values are good candidates for inclusion in a minimally working interpreter because they are generally the easiest forms of data and operations to implement.

Another form of primitive operation, often used in statement-oriented languages like C, is referred to as the "sequence operator", and it's what is meant by the semicolon in the the following C program fragment:

```
1;
2;
```

which produces the following AST:



where ε is shorthand for "no operation." In any case, choose one operation that makes sense in *your* langauge.

Minimally Working Interpreter

The following constitutes a "minimally working interpreter":

(a) Your AST can represent at least one kind of data.

(b) Your AST can represent at least one operation.

(c) Your parser can recognize a program consisting of your one kind of data and your one operation and it produces the appropriate AST.

(d) Your evaluator can evaluate your one operation using operands consisting of your data, and if necessary, expressions consisting of your data and operation. In other words, it can recursively evaluate subexpressions, where appropriate. Note that it is important that your minimally working interpreter *do something*, whether that be to compute a value, or write to a file, etc.

Minimal Formal Grammar

Additionally, you should update your specification with a *formal* definition of the minimal grammar. For example, if our minimal working version is a scientific calculator that only supports addition, our first pass on the grammar might be:

```
<expr>   ::= <number><ws><op><ws><expr>
           |  <number>
<number> ::= <d><number>
           |  <d>
<d>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>     ::= +
<ws>     ::= ␣ | ε
```

Where ␣ denotes a space character and ε denotes the empty string. Note that we did not write the following similar grammar.

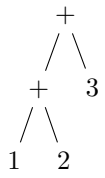
```
<expr>   ::= <expr><ws><op><ws><expr>
           | <number>
<number> ::= <d><number>
           | <d>
<d>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>     ::= +
<ws>     ::= ␣ | ε
```

The reason is that this latter grammar is what we call *left recursive*. In particular, the production `<expr><ws><op><ws><expr>` is problematic for mechanical reasons: when we convert our BNF into a program (a parser), it is possible to construct a program that recursively expands the left `<expr>` infinitely without ever consuming any input. When using recursive descent parsers such a parser combinators, we must be careful to ensure that recursive parsers always consume some input on each step, otherwise, we run the very real danger of our parser getting stuck in an infinite loop. If your grammar is left recursive, you should redesign it so that it is no longer left-recursive.

Since your grammar only has a single operation, precedence will not yet be an issue. However, if you can include more than one such operation, you will need to think about the associativity of your operator. Is it left or right associative? For example, addition is typically left associative, therefore the following expression

$$1 + 2 + 3$$

should produce the following AST



Be sure to explain your operator's associativity in the next section.

Minimal Semantics

Finally, you should explain the semantics of your data and operators. For example, you might build the following table.

Syntax	Abstract Syntax	Type	Prec./Assoc.	Meaning
n	`Number of int`	`int`	n/a	n is a primitive. We represent integers using the 32-bit F# integer data type (`Int32`).
$e_1 + e_1$	`PlusOp of Expr * Expr`	`int -> int -> int`	1/left	`PlusOp` evaluates $e1$ and $e2$, adding their results, finally yielding an integer. Both e_1 and e_1 must evaluate to `int`, otherwise the interpreter aborts the computation and alerts the user of the error.