Homework 9

Due Thursday, Nov 15 by 10pm

Turn-In Instructions

For text response questions, 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.

For each coding question in this assignment, create a project directory. You should be able to `cd` into this directory and then run the program by typing the command "`dotnet run`". Each program should be split into two pieces: a "`Program.fs`" file that contains the `main` method and associated program-startup routines (like argument parsers), and another "`Library.fs`" file that contains the function(s) of interest in the question. All library code should be in a module named "`CS334`". Be sure to provide usage output (defined in `main`) for all programs that require arguments. For full credit, your program should both build and run correctly.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw9_<your user name>`. For example, my repository would be `cs334hw9_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.

This assignment is due on Thursday, Nov 15 by 10pm.

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:

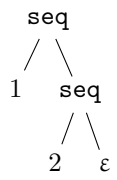
```
$ dotnet run "1 + 2"
3
```

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.

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.

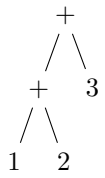
```
<expr>   ::= <expr><ws><op><ws><expr>
          |  <number>
<number> ::= <d><number>
          |  <d>
<d>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>     ::= +
<ws>     ::= ␣ | ε
```

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 refactor 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, you will need to think about the associativity of your operator. Is it left or right associative? For instance, 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.

Deliverables

Supply your updated specification as a LaTeX file and call it "q1.tex". Also provide a PDF for your LaTeX.

The project directory for your minimally working interpreter should be called "q1".

Q2. (*30 points*) ..... Beta Normal Form

This question asks you to write F# code that performs beta reduction on lambda calculus expressions.

Recall that beta reduction is the process of substituting a bound variable with a value "applied to" a lambda abstraction. This form of substitution, when combined with alpha reduction from your last assignment, is sufficient to carry out any computation that we can do in principle. Therefore, investigating how beta reduction works, in a mechanical way, is helpful for understanding how one might implement other features in a real programming language. For example, variable and function definitions, function calls, and so on, can be implemeted as syntactic sugar on top of the lambda calculus.

Your end goal is to write the following function, which reduces an arbitrary lambda expression to *beta normal form*,

```
betanorm : e:Expr -> Expr
```

where `e` is a lambda expression, and the function returns an expression in beta normal form. In a beta normalized expression, there is no more "work" left to do: all of the applications that have lambda abstractions on their left sides have been beta reduced.

For example, given the following lambda calculus expression written in our expression language,

```
((Lx.x)y)
```

the `betanorm` function returns the expression `y`. The intuition is that `betanorm` "rewrites" an AST representing the lambda calculus in much the same way a human would rewrite a lambda expression on paper.

We use the same AST data types as in your parsing and alpha normalization assignments,

```
type Expr =
| Variable of char
| Abstraction of char * Expr
| Application of Expr * Expr
```

therefore, you should allow the user to input lambda expressions as plain text, using a parser. You will be supplied with starter code for a complete lambda expression parser and an alpha reduction implementation, but you may use your own code if you wish.

We will begin by decomposing this problem into a small set of helper functions in order to write the `betanorm` function.

(a) Write the function

```
sub : v:char -> with_e:Expr -> in_e:Expr -> Expr
```

that performs a substitution where `v` is a variable bound in a lambda abstraction, `with_e` is an argument being applied to a lambda abstraction, and `in_e` is the body of a lambda abstraction. For example, given the expression,

```
((Lx.(xx))y)
```

the parts are labeled as follows:

Subexpression	Name
`x`	`v`
`y`	`with_e`
`(xx)`	`in_e`

and the result of calling `sub` with these arguments should be the expression `(yy)`.

To be precise, the `sub` function carries out the substitution operation implied by the beta-equivalence rule:

$$(\lambda x.\texttt{<expr>})y =_\beta \texttt{[y/x]<expr>}$$

In other words, we eliminate the lambda abstraction by replacing every occurrence of `v` with `with_e` in the expression `in_e`.

Note that, for simplicity, you may assume that the expression `e` is already in alpha-normal form. Finally, `sub` should be recursive, continuing to replace `v with_e in_e` for as long as it makes sense.

(b) Write the function

```
betastep : e:Expr -> Expr
```

that performs, at most, one step of beta reduction on the expression `e`.

The logic of the function should depend on the case of the `e: Expr` ADT.

- `Variable v`: Return e.
- `Abstraction(v,e')`: Perform one step of beta reduction on the expression `e'` and return a new `Abstraction` of `v` and the one-step-beta-normalized `e'`.
- `Application(e1, e2)`: The result depends on the type of `e1`.
  - `Variable`: Perform one step of beta reduction to `e2` and return a new `Application`, where `e1` is the left expression and the one-step-beta-normalized `e2` is the right expression.
  - `Abstraction(e1v,e1e)`: Substitute `e1v` with `e2` in `e1e`.
  - `Application`: First try beta reducing `e1`. If `e1` reducible, return a new `Application`, where the one-step-beta-normalized `e1` is the left expression and `e2` is the right expression. Otherwise, if `e1` is not reducible, try beta reducing `e2`. If `e2` is reducible, return a new `Application`, where `e1` is the left expression and the one-step-beta-normalized `e2` is the right expression. Otherwise, if neither expression can be reduced, return `e`.

A useful property of the lambda calculus is the identity

```
e = betastep e
```

where `e` is an expression in beta-normal form. In other words, beta reducing a beta-normalized expression yields the original expression.

(c) Finally, write the function,

```
betanorm : e:Expr -> Expr
```

where `e` is a lambda expression, and the function returns an expression in beta-normal form. `betanorm` should first alpha-normalize `e` to ensure that there are no name conflicts, and then it should perform one step of beta normalization. If the result of one step of beta normalization is an expression in beta normal form, then `betanorm` should return the expression, otherwise, it should call itself recursively. In other words, `betanorm` performs as many steps of beta normalization as are necessary to beta normalize the entire expression.

Here are some sample lambda expressions and their conversion to beta-normal form.

`(Lx.x)`	`(Lx.x)`
`(Lx.(Lx.x))`	`(Lx.(La.a))`
`(((Lx.x)(Lx.(xx)))(Lx.(xa)))`	`(aa)`
`(((Lx.x)(Ly.(yy)))(Lz.(za)))`	`(aa)`
`(((Lx.(xx))(Lx.(yx)))z)`	`((y(Lb.(yb)))z)`
`((Lx.((Ly.(xy))y))x)`	`(xy)`

(d) Be sure to document the above functions using XML comment blocks.

(e) The project directory for this question should be called "`q2`". You should be able to run your program on the command line by typing, for example, "`dotnet run "(Lx.(Lx.x))"`" and the program's output should be the beta-normalized expression pretty-printed using `lambdaprint`.

Q3. (*10 points*) ..... Bonus: Lambda Reduction Prover

Modify your beta reduction program to print step-by-step beta reduction proofs. Your program should not print reduction proofs as a side effect. Instead, you should write a new top-level function `reduce : e:Expr -> (Expr*Reduction) list` that generates a lambda reduction proof as a list of lambda expressions. `Reduction` is a new ADT of your own design that explains the reduction carried out in each step (e.g., informally, "Beta-reduce `x` with `y` in `(yy)`"). Note that, to produce the extra information needed to support the `reduce` function, you will likely need to rewrite your alpha and beta reduction functions.

Your top-level `main` function should then print out the proof. For example, given the expression `(((Lx.x)(Lx.(xx)))(Lx.(xa)))`, your `main` function should produce a proof that looks like:

```
(((Lx.x)(Lx.(xx)))(Lx.(xa))) | given
(((Lx.x)(Lb.(bb)))(Lc.(ca))) | alpha reduce x with b, x with c
((Lb.(bb))(Lc.(ca)))         | beta reduce x with (Lb.(bb)) in x
((Lc.(ca))(Lc.(ca)))         | beta reduce b with (Lc.(ca)) in (bb)
((Lc.(ca))(Lb.(ba)))         | alpha reduce c with b
((Lb.(ba))a)                 | beta reduce c with (Lb.(ba)) in (ca)
(aa)                         | beta reduce b with a in (ba)
(aa)                         | done
```

For full credit, your proof needs to be neatly indented as shown above.

The project directory for this question should be called "`q3`". You should be able to run your program on the command line by typing, for example, "`dotnet run "(Lx.(Lx.x))"`" and the program's output should be a reduction proof like the one shown above.