Lab 9

Due Sunday, November 19 by 10:00pm

Handout 25

CSCI 334: Fall 2023

Turn-In Instructions

In this lab, you will begin implementing your programming language. Be sure to follow the instructions below for organizing your language project. You will use this repository for your project for the remainder of the semester.

Turn in your work using the `git` repository assigned to you. The name of the repository will have the form `https://aslan.barowy.net/cs334-f23/cs334-project-<USERNAME1>-<USERNAME2>.git`. For example, if your username is `abc1` and your partner's is `def2`, the repository would be `https://aslan.barowy.net/cs334-f23/cs334-project-abc1-def2.git`.

Honor Code

This is a pair programming lab. Like previous partner labs, you may work with a partner. Unlike previous labs, you may collaborate to produce a single solution. You do not need to submit a `collaborators.txt` file for this lab.

This assignment is due on Sunday, November 19 by 10:00pm.

Sanity Check: Students sometimes submit incomplete assignments, accidentally forgetting to run `git add` for all of their files. Fortunately, there is an easy way to make sure that this does not happen to you. Before you are done, `git clone` your repository to a new folder and then try building/running everything. It only takes a couple minutes and can spare you from headaches later on.

Reading

1. **(Read)** Read "Evaluation" from the course packet.
2. **(Read)** Read "Implementing Variables" from the course packet.

Problems

Q1. (90 points) .... Minimal Project Prototype

For this assignment, you will build a minimally working version of your language. You should also create a project specification document that describes your minimally working version. Please put your code in the `code` folder and your specification in the `docs` folder.

A minimally working interpreter has the following components:

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

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

(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 "combining forms" to implement.

Another form of combining form, 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:

```
   seq
   / \
  1   seq
      / \
     2   ϵ
```

where ϵ is shorthand for "no operation." In any case, choose one combining form 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 combining form.
(c) Your parser can recognize a program consisting of your one kind of data and your one combining form and it produces the appropriate AST.
(d) Your evaluator can evaluate your one combining form using operands consisting of your data, and if necessary, expressions consisting of your data and combining form. 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 combining form, precedence will not yet be an issue. However, if you can include more than one such combining form, you will need to think about the associativity of your combining form. 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

```
      +
     / \
    +   3
   / \
  1   2
```

Remember that you can use the `forest` page for drawing trees.

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.

Your semantics does not need to be formal, and it does not need to be in a table, but it should discuss the items shown the table above.