Lab 10

Due Wednesday, April 30 by 11:59pm

Handout 22
CSCI 334: Spring 2025

Turn-In Instructions

For this lab, you will checkout a project repository that you will continue to use for the rest of the semester. Be sure to follow the instructions for committing your work to the appropriate branch.

Turn in your work using your assigned `git` repository. The name of your repository will have the form `https://aslan.barowy.net/cs334-s25/cs334-project-<USERNAME1>-<USERNAME2>.git` For example, if your CS username is `abc1` and your partner's is `def2`, the repository would be `https://aslan.barowy.net/cs334-s25/cs334-project-abc1-def2.git`

Pair Programming Assignment

This is a pair programming lab. As with previous partner labs, you may work with a partner. However, for a pair programming assignment like this one, 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 Wednesday, April 30 by 11:59pm.

Reading

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

Problems

Q1. (15 points) ..... Set the Project Branch

Your work must be committed to a branch called `prototype`.

To create and switch to a `prototype` branch:

(a) Run `git checkout -b prototype`, which will create a new branch called `prototype`.

(b) Make your changes, then `git add` and `git commit` as appropriate to save your changes.

(c) To push the new branch to `aslan` for the first time, run `git push -u origin prototype`. We need to push differently than usual because the `prototype` branch you just created does not exist on the server. Subsequent calls to `git push` can be made as usual.

(d) Go to your repository and verify that your new `prototype` branch appears in the web interface.

After you have pushed the `prototype` branch to the server, if your partner wants to check out the same branch, they should first `git pull` and then run `git checkout prototype`.

Q2. (85 points) ..... Minimal Project Prototype

For this question, you will build a minimally working version of your language. You will also update your project specification document as you design syntax to support your minimally working version. Put your code in the `code` folder and update 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 abstract syntax tree. Put your AST in a library file called `AST.fs`. The namespace for the interpreter should also be called `AST`.

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

(d) A driver program. The driver code, `Program.fs`, should contain a `main` function that takes input from the user, parses, and interprets it using the appropriate library calls, and displays the result. 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 containing same program, 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.

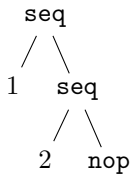
```
$ dotnet run
Usage:
  dotnet run <file.calc>

  Calculang will frobulate your foobars.
```

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:



where `nop` is an AST node that does "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 any AST produced by your parser. In other words, it may recursively evaluate subexpressions when 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. If this part is confusing or unclear, please come speak with me.

(e) The entire language can be invoked on the command line as described above.

The goal here is not to implement your entire language. Instead, find a tiny seed of your language and plant it. By the end of the semester, you will grow that tiny seed into the language you envision.

Do as much as you need to convince yourself that you're on the right track. If you need to cut corners at this point, that's OK. The most important thing is that your language has all three parts (parser, AST, evaluator) in some form, and that those parts work, even though they may not be fully functional.

Minimal Formal Grammar

Finally, update the Syntax section of your specification with a formal definition of your minimal grammar. For example, if our minimal working version were a scientific calculator that only supports addition, our first pass on the grammar might be:

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

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

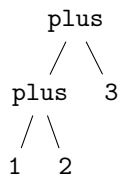
```
<expr>   ::= <expr>␣<op>␣<expr>
           |  <number>
<number> ::= <d><number>
           |  <d>
<d>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>     ::= +
```

The reason is that this latter grammar is what we call *left recursive*. In particular, the production `<expr>␣<op>␣<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 input, otherwise, we run the very real danger of our parser getting stuck in an infinite loop. If your grammar is left recursive, 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



How to Organize

Your submission should organize your files into two different directories. Your language implementation should be placed in a directory called `code`, and as usual, it must be written in F#. Your draft specification must be placed in a directory called `docs`. Since we are updating our specification from the last project checkpoint, you may start by copying your old LaTeX files into the `docs` directory.

After you `git push` your project, you should immediately create a new branch called `final` so that future project work goes into a different branch. Just follow the branching instructions above again, substituting `final` where you see `prototype`.