Lab 9

Due Monday, April 29 by 11:59

Handout 25
CSCI 334: Spring 2024

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 `git` repository will have the form `https://aslan.barowy.net/cs334-s24/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-s24/cs334-project-abc1-def2.git`

You should have received an invite to commit to the repository via email. If you did not receive an email, please contact me right away!

Pair Programming Assignment

This is a pair programming lab. Like previous partner labs, you may work with a partner. However, for pair programming assignments, 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 Monday, April 29 by 11:59.

Reading

1. **(Read)** Read "Evaluation" from the course packet.
2. **(Read)** Read "Appendix B: Branching in `git`" from the course packet.

Problems

Q1. (10 points) ..... Set the Project Branch

We will be using a `git` feature that helps you juggle multiple lines of development in a single `git` repository. As you know, `git` commits are a part of a commit history called the "`git` log." By default, you commit to a history called `main`. What you may not know is that a `git` repository can contain multiple commit histories, called *branches*. All commit histories must be related to the `main` branch in some way. The entirety of all commits across all histories forms a graph, specifically, a directed acyclic graph (DAG). In the real world, branches are frequently used to experiment with changes that might otherwise be disruptive to others working on "mainline" code.

For this lab, create a new branch called `prototype`. See the reading "Appendix B: Branching in `git`" for instructions. After you create a local branch, make some commits, and then try to push it to the class `git` server, the push will fail. For example,

```
$ git push
fatal: The current branch prototype has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin prototype

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

Observe that the `git` message above says `fatal`, meaning that the push did not succeed. The reason is that you have to explicitly tell `git` to track your new branch the first time you push. Fortunately, git tells us exactly what to type in the message above.

```
$ git push --set-upstream origin prototype
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 290 bytes | 290.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To aslan.barowy.net:cs334-s24/cs334-lab09-dwb1.git
 * [new branch]      prototype -> prototype
branch 'prototype' set up to track 'origin/prototype'.
```

This push succeeded and it tells us that the local branch `prototype` is now connected with the server's branch `origin/prototype`. From now on, you can type `git push` as usual.

Finally, be aware that when any two developers work on the same branch, it is always possible that `git` will not be able to automatically "merge" the two sets of changes. This common occurrence is called a "merge conflict." When you see this, keep in mind that it is a safety feature that prevents you from accidentally overwriting another developer's changes. It is only frustrating when you don't know that conflicts are normal. You should expect merge conflicts to happen, and in a large organization, they will happen with some frequency. The reading walks you through handling them.

Q2. (40 points) ..... Project Description

For this question, you will start describing your final project: a programming language of your own design.

Many programming language specifications begin as informal documents, and this is the template that we will follow in this class. With each stage of your project, you will revisit this document, making it clearer and more precise as you work. It will be a "living document." By the end of the semester, your final specification will include a formal syntax and an informal, but precise, description of your language's semantics. It should clearly document your software artifact, which will be an interpreter for the language. For now, we will start informally, and you should feel free to borrow text from your brainstorm.

Goals

To help you see where we are going, here are the goals of the final project. Your language need not be (and I discourage you from trying to build) a Turing-complete programming language. Instead, focus on solving a single kind of problem. In other words, design a *domain-specific programming language.*

By the end of the semester, your project must achieve all of the following objectives:

(a) It should have a grammar capable of expressing either an infinite or practically-infinite number of possible programs.

(b) It should have a parser that recognizes a syntatically-correct program, outputting the corresponding abstract syntax tree.

(c) It should have an evaluator capable of interpreting any valid AST.

(d) The language should do some computational work.

If it is convenient to solve your problem by reducing to or extending a lambda calculus interpreter, you may propose such a solution. However, in most cases, it will likely be easier to design a purpose-built interpreter.

Structure of the Specification

Version 1.0 of your specification should explicitly include the following sections. The purpose of this document is to convince yourself that your language implementation is possible. If you aren't convinced, add more detail to convince yourself!

(a) Introduction

2+ paragraphs. What problem does your language solve? What makes you think that this problem should have its own programming language?

(b) Design Principles

1+ paragraphs. Languages can solve problems in many ways. What are the aesthetic or technical ideas that guide its design?

(c) Examples

3+ examples. Keeping in mind that your syntax is still informal, sketch out 3 or more sample programs in your language.

(d) Language Concepts

1+ paragraphs. What are the core concepts a user needs to understand in order to write programs? Think in terms of both "primitives" and "combining forms." What are the key ideas and how are they combined?

(e) Syntax

This section of your specification should begin to describe your syntax formally, using BNF. You will write this text as a part of the next section of the lab handout (minimal project prototype).

(f) Semantics

As much as is needed. How is your program interpreted? This need not be formal yet, however, you should demonstrate that you've thought about how your program will be represented and evaluated on a computer. It should answer the following questions.

i. What are the primitive kinds of values in your system? For example, a primitive might be a number, a string, a shape, or a sound. Every primitive should be an idea that a user can explicitly state in a program written in your language.

ii. What are the combining forms in your language? In other words, how are values combined in a program? For example, your system might combine primitive "numbers" using an operation like "plus." Or perhaps a user can arrange primitive "notes" within a "sequence."

iii. How is your program evaluated? In particular,

A. Do programs in your language read any input?

B. What is the effect (output) of evaluating a program? Does the language produce a file or print something to the screen? Use one of your example programs to illustrate what you expect as output.

How to Organize

The project directory for this question should be called `docs`. You must use LaTeX for your specification.

Q3. (50 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 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 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.

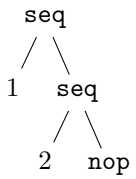
```
$ dotnet run
Usage:
  dotnet run <file.calc>

  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.

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.

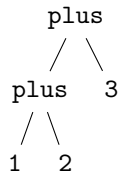
```
<expr>   ::= <expr>␣<op>␣<expr>
          |  <number>
<number> ::= <d><number>
          |  <d>
<d>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>     ::= +
```

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

The project directory for this question should be called "`code`". You must use F# for your implementation.