

Homework 9

Due Sunday, November 13 by 10:00pm

Handout 20
CSCI 334: Fall 2022

Turn-In Instructions

For the coding question in this assignment, create a project in the “q1” directory. You should be able to 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.

For the reading response questions, use the \LaTeX template found in the “q2” directory. The template should compile successfully without modification. Please note that for full credit, you must submit both your `.tex` source files as well as the rendered `.pdf` file. Your source file should be called `lab-9.tex` and your PDFs should be called `lab-9.pdf`. (5 points)

Turn in your work using the Gitlab repository assigned to you. The name of the Github repository will have the form `https://evolene.cs.williams.edu/cs334-f22/<YOUR_USERNAME>/lab09.git`. For example, if your CS username is `22abc1`, the repository would be `https://evolene.cs.williams.edu/cs334-f22/22abc1/lab09.git`.

Honor Code

This is a partner lab. You may work with another classmate if you wish, and you may co-develop solutions. Remember: although you can work on code together, you must each independently write up and submit your solution. No code copying is allowed. **Be sure to tell me who your partner is** by committing a `collaborators.txt` file to your repository (5 points). Be sure to include this file even if you work by yourself.

This assignment is due on Sunday, November 13 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. (Required) “Parser Combinators”
2. (Required) “How to Fix a Motorcycle”

Problems

Q1. (50 points) Parsing with Combinators

(a) Given the following grammar in Backus-Naur Form,

```
<expr> ::= <var>
        | <abs>
        | <app>
<var>  ::=  $\alpha \in \{a \dots z\}$ 
<abs>  ::= (L<var>.<expr>)
<app>  ::= (<expr><expr>)
```

and the algebraic data type,

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

provide an implementation for the parser function

```
parse(s: string) : Expr option
```

In other words, given a `string s` representing a valid expression, `parse` should return `Some` abstract syntax tree (the `Expr`) of the expression. When given a `string s` that is not a valid expression, `parse` should return `None`.

You may use any of the combinator functions defined in the assigned reading on parser combinators in your solution. You may find the `pletter`, `pchar`, `pstr`, `pseq`, `pbetween`, `pleft`, `peof`, `<|>`, and `|>>` combinators to be the most useful.

Note that because your expression parser will be recursive, we need to do a little bookkeeping to keep F# happy. The first parser that appears your implementation should be written like:

```
let expr, exprImpl = recparser()
```

`recparser` defines two things: a declaration for a parser called `expr` and an implementation for that same parser called `exprImpl`. Later, once you have defined all of the parsers that `expr` depends on, write:

```
exprImpl := (* your expr parser implementation here *)
```

Note: use of `recparser` is admittedly a little bit of a hack. We use it because F# requires us to define functions before we use them, which means that F# gets unhappy when parsers are defined recursively. `recparser` is one way around this problem. You should only need to use `recparser` once in this problem. To be *crystal clear*, your code should probably have at least the following definitions in it:

```
let expr, exprImpl = recparser()
let variable : Parser<Expr> = (* variable parser implementation *)
let abstraction : Parser<Expr> = (* abstraction parser implementation *)
let application : Parser<Expr> = (* application parser implementation *)
exprImpl := (* expr parser implementation *)
```

You are strongly encouraged to define additional parser helpers if they make the problem simpler for you to solve.

- (b) Provide a function `prettyprint(e: Expr) : string` that turns an abstract syntax tree into a string. For example, the program fragment

```
let asto = parse "((Lx.x)(Lx.y))"  
match asto with  
| Some ast -> printfn "%A" (prettyprint ast)  
| None     -> printfn "Invalid program."
```

should print the string

```
Application(Abstraction(Variable(x), Variable(x)), Abstraction(Variable(x), Variable(y)))
```

- (c) Be sure to document all of your functions using comment blocks.
- (d) (Optional) For a fun challenge, extend your implementation to do one or more of the following:
- Accepts arbitrary amounts of whitespace between elements.
 - Accepts the λ character in addition to L for abstractions.
 - Allows the user to omit parens when the lambda calculus' rules of precedence and associativity allow the expression to be parsed unambiguously.

If you decide to tackle any of the above, be sure to tell us that you attempted a bonus.

The last item is challenging, but it is quite satisfying to produce a parser that can read in expression just as they are written in the course packet. If you're looking to push yourself, give it a try!

Suggestion: Parser combinators are an elegant and conceptually simple way to develop parsing algorithms. However, parsing text is never easy, because machines read input very strictly, unlike humans. Therefore, the operation of a parser is frequently counterintuitive. Unfortunately, combinators do not play nicely with breakpoint debuggers like the one found in Visual Studio Code. You are strongly encouraged to use the `<!>` “debug parser” along with the `debug` combinator. To debug, use `debug` instead of `prepare`. Better yet, use a `DEBUG` flag to choose which function to call, like:

```
let i = if DEBUG then debug input else prepare input
```

The project directory for this question should be called “q1”. You should be able to run your program on the command line by typing, for example, “`dotnet run "((Lx.x)(Lx.y))"`” and output like the kind shown above should be printed to the screen.

Q2. (40 points) How to Fix a Motorcycle

Read the excerpt of Zen and the Art of Motorcycle Maintenance titled “How to Fix a Motorcycle” by Robert Pirsig, and then answer the following questions. Please try to keep your responses short. For all three questions below, you should write between 200 and 400 words total.

- Think back about your experiences doing programming in the past. Think of a time when it was difficult and you were feeling demoralized. This may have been a programming assignment, programming for work, or programming for fun. Now that you've read Pirsig's essay, do any of his “gumption traps” apply to your experience? Describe your experience.
- Whether Pirsig's advice applies to your situation or not, what would you do differently now if you were in that situation again?
- To what extent do you think your situation could have been improved by a better programming language? In other words, suppose your programming language helped you more. What pitfalls or gumption traps might be avoided? Don't worry too much about whether your imagined features are impractical to implement.

Q3. ($\frac{1}{10}$ bonus point) Optional: Feedback

I always appreciate hearing back about how easy or difficult an assignment is.

For $\frac{1}{10}$ of a bonus to your final grade, please fill out the following Google Form.