Lab 7

Due Sunday, November 5 by 10:00pm

Handout 21
CSCI 334: Fall 2023

Turn-In Instructions

Part of this assignment must be written using LaTeX. I provide a LaTeX template in your repository for you to get started. The template I provide compiles without error as-is. For full credit, you must submit both your `.tex` source file as well as the rendered `.pdf` file. Your source file should be called `lab-7.tex` and your PDF should be called `lab-7.pdf`. (5 points)

For each programming question in this assignment, create a project directory. For example, the source directory for question 1 should be in a folder called "`q1`". You should be able to `cd` into this directory and then run the program by typing the command "`dotnet run`", with additional arguments depending on the question.

Each program should be split into two pieces: a "`Program.fs`" file that contains the `main` method and associated program-startup helpers (if needed), 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`". For full credit, your program should both build and run correctly. Be sure to provide usage output (defined in `main`) for all programs that require arguments, and be sure to validate the input that the user gives you.

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-lab07-<USERNAME>.git`. For example, if your username is `abc1`, the repository would be `https://aslan.barowy.net/cs334-f23/cs334-lab07-abc1.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. Be sure to commit this file whether you worked with a partner or not. If you worked by yourself, `collaborators.txt` should contain something like "I worked by myself." (5 points)

This assignment is due on Sunday, November 5 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)** "Parsing"
2. **(As needed)** Refer to previous readings on F#.

Problems

Q1. (45 points) ..... Parsing with Combinators

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

```
<expr> ::= <var>
         |  <abs>
         |  <app>
<var>  ::= α ∈ {a...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 in 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 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 *)
```

Please do define additional parsers if they help you solve the problem.

(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) (Bonus) For an optional challenge, extend your implementation to do one or more of the following:

i. Accepts arbitrary amounts of whitespace between elements.

ii. Accepts the λ character in addition to `L` for abstractions.

iii. 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 in a comment at the top of `Program.fs`.

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` function. To debug, use `debug` instead of `prepare`. Better yet, use a `DEBUG` flag, which you can toggle on and off as needed, to choose which function to call:

```
let DEBUG = true
// ... your code ...
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. (5 points) ..... Project Partner

If you plan to collaborate with a classmate on your final project, please let me know who that person is by filling out the following project partner sign-up sheet (`https://forms.gle/KAipoaP4BVnoiCLp6`).

Q3. (40 points) ..... Project Brainstorming

For your final project, you will design and implement a programming language. For this part of the assignment, you should think of three different possible final projects you might explore. Two of those proposals should involve artworks at WCMA's Object Lab. The third can be any project that you want. Feel free to let your imagination run wild.

If you find the notion of creating your own programming language daunting, don't worry. For future project checkpoints, you are welcome to fall back to a "default" Object Lab project whose scope will be a little less open-ended. For now, throw caution to the wind with the assurance that at this stage any idea you dream up is non-binding.

Note that a programming language need not be a so-called general purpose programming language. A general purpose programming language is equivalent in expressive power to the lambda calculus or a

Turing machine, meaning that is can be used to compute anything. Instead, I encourage you to explore domain specific programming languages, or DSLs. DSLs are usually tailored toward solving a specific problem.

You have probably used a DSL without even realizing it. Some example DSLs are:

(a) `make`
(b) GraphViz
(c) Hypertext Markup Language (HTML)
(d) Extensible Markup Language (XML)
(e) Structured Query Language (SQL)
(f) Markdown
(g) LaTeX
(h) Scalable Vector Graphics (SVG)
(i) JavaScript Object Notation (JSON)

If you have a personal itch, scratch it. For example, I often have to grade student work, and the part I always mess up involves computing scores. Therefore, the grading rubrics I supply to my teaching assistants are actually written in a domain specific language I designed called `tabulator`.

Be creative! Do you love music, or art, or literature? Can you make a language that generates music? Could you make a language that creates art? Could you make a language that generates poetry? Some of your former classmates have designed languages that have done precisely these things and more.

For each potential language, describe

(a) The purpose of the language. What problem does it solve? When I have trouble focusing on this part, it often helps me to think in terms of *inputs* and *outputs*.
    i. *Programs*, written in the language you design, are the inputs.
    ii. The output is what happens *when one of those programs is evaluated*.
(b) Write two sample programs to demonstrate what the language might look like. Feel free not to be constrained by reality at this point. Just imagine that your programming language already exists.

For the two Object Lab pieces, you can propose anything computational involving that artwork. If you are struggling with this idea, as a "default project," consider the following purpose: your language should generate artwork in the style of the artist. One way to do this would be to either make your program non-deterministic (it "interprets" the program differently every time), or perhaps it takes an input that tells the interpreter how to proceed with variations deterministically (e.g., by specifying a random seed).

For example, for the Josef Albers painting, "Homage to the Square: Warming," one kind of "fantasy program" might be:

```
5 squares, from gray to orange, stacked and 25% smaller, variation 1.
```

With small tweaks to the same syntax, we can also get very different outputs.

```
8 squares, from gray to blue, side-by-side and equal size, variation 2.
```

(We will build an "Albers interpreter" together in class, so you cannot propose this particular language.)

Observe that for this language is it easy to ask the following questions: What's the grammar? Does it take any additional inputs? What are the outputs? Think about those questions; we will answer them in a future lab. If the answers come easy, it means it's probably an interesting and well-defined

language. At a future stage, you will say precisely how your language is parsed and interpreted, so although you don't have to write anything down now, you might spend a few minutes thinking about it.

Be sure to submit your project brainstorm in a file called "`lab-7.tex`" along with a pre-built "`lab-7.pdf`" in a folder called "`project`".

Q4. ($\frac{1}{10}^{\text{th}}$ bonus point) ..... Optional: Feedback

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

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