Lab 8

Due Monday, April 22 by 11:59pm

Handout 23
CSCI 334: Spring 2024

Coding Guidelines

Each question in this assignment should go into the appropriate project directory. For example, the solution to question 1 should be in a folder called "`q1`". When a solution is a program, one should be able to `cd` into the question directory and then run your program by typing the command "`dotnet run`", with additional arguments depending on the question.

Every 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. Library code should be contained within 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.

If any of your programs take input from the user, be sure that your program validates input: when a user fails to supply input, or supplies input that does not make sense, your program should print a usage message and return with a nonzero exit code. Users should never experience a program crash in this class; exceptions should be prevented from arising or be caught whenever bad input is encountered. Think through problem corner cases carefully.

Turn-In Instructions

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-lab08-<USERNAME>.git`. For example, if your CS username is `abc1`, the repository would be `https://aslan.barowy.net/cs334-s24/cs334-lab08-abc1.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!

Group Programming Assignment

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. 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 Monday, April 22 by 11:59pm.

Reading

1. **(Required)** "Parser Combinators"
2. **(As Needed)** "Previous readings on F#"

Problems

Q1. (95 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 F# requires us to define functions before we use them. If you look at the grammar for the lambda calculus, you will see an obstacle: `<expr>` depends on the definition for `<app>`, and `<app>` depends on the defintion for `<expr>`. Therefore, we cannot implement parsers for those nonterminals without running afoul of F#'s rules. The `recparser` function is one way around this chicken-and-egg kind of problem. It allows us to separate the declaration of a parser from its definition. You should only need to use `recparser` once in this problem.

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 *)
```

To be *crystal clear*, your code should probably have at least the following definitions in it:

```
let expr, exprImpl             = recparser() (* declares that an expr parser exists *)
let variable    : Parser<Expr> = ...         (* defines a variable parser *)
let abstraction : Parser<Expr> = ...         (* defines an abstraction parser *)
let application : Parser<Expr> = ...         (* defines an application parser *)
exprImpl                       := ...        (* defines the expr parser *)
```

You may define as many additional parsers as you feel will help you.

(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 me** 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 it is 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. ($\frac{1}{10}^{\text{th}}$ 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.