Lab 8

Due Wednesday, April 16 by 11:59pm

Handout 19
CSCI 334: Spring 2025

Coding Guidelines

In this lab, you will be working on a single, large program. This program should be developed within the root directory of your repository. For full credit, your program must compile and run correctly.

As usual, your program will be run by using the `dotnet run` command. When mandatory arguments are omitted, or when they don't make sense, your program should provide usage output and exit 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 your assigned `git` repository. The name of your repository will have the form `https://aslan.barowy.net/cs334-s25/cs334-lab08-<USERNAME>.git`. For example, if your CS username is `abc1`, the repository would be `https://aslan.barowy.net/cs334-s25/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 Wednesday, April 16 by 11:59pm.

Reading

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

Problems

Q1. (95 points) ..... Parsing with Combinators

JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. JSON defines a small set of formatting rules for the portable representation of structured data. JSON is the de-facto data interchange format used on the web. In this lab, you will implement a JSON parser in F# using parser combinators.

(a) The following grammar in Backus-Naur Form represents the JSON format,

```
<json>        ::= <number> | <string> | <boolean> | <list> | <object>
<number>      ::= n ∈ ℤ
<string>      ::= " <ltr>* "
<ltr>         ::= A .. Z | a .. z | 0 .. 9 | ␣ | <symbol>
<symbol>      ::= , | . | ~ | ! | ? | @ | # | $ | % | ^ | & | * | ( | ) | - | + | _ | =
<boolean>     ::= true | false
<list>        ::= [ <json> <more_list> ]
<more_list>   ::= ϵ | , <json> <more_list>
<object>      ::= { <field> <more_object> }
<more_object> ::= ϵ | , <field> <more_object>
<field>       ::= <string> : <json>
```

where `..` represents a range of characters, ␣ represents a whitespace character, and ϵ represents the empty string.

The following algebraic data type represents a JSON abstract syntax tree in F#.

```
type JSON =
| JNumber of int
| JString of string
| JBoolean of bool
| JList of JSON list
| JObject of (JSON * JSON) list
```

Provide an implementation for the parser function

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

In other words, given a `string s` representing valid JSON, `parse` should return `Some` AST. When given a `string s` that is not valid JSON, `parse` should return `None`.

You may use any of the combinator functions defined in the assigned reading on parser combinators in your solution. You will likely need to use the `pdigit`, `pletter`, `pchar`, `pstr`, `pseq`, `pbetween`, `pleft`, `pright`, `peof`, `pmany0`, `pmany1`, `<|>`, and `|>>` combinators. You will also likely need the `stringify`, `int`, and `id` helper functions.

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: `<json>` depends on the definition for `<list>`, and `<list>` depends on the defintion for `<json>`. 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 json, jsonImpl = recparser()
```

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

```
jsonImpl := (* your json parser implementation here *)
```

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

```
let json, jsonImpl             = recparser() (* declares that a json parser exists *)
let number      : Parser<JSON> = ...         (* defines a number parser *)
let strng       : Parser<JSON> = ...         (* defines a string parser *)
let boolean     : Parser<JSON> = ...         (* defines a boolean parser *)
let list        : Parser<JSON> = ...         (* defines a list parser *)
let object      : Parser<JSON> = ...         (* defines an object parser *)
jsonImpl                       := ...        (* defines the json parser *)
```

however you will likely need to define some helper parsers along the way. Note that the above intentionally misspells `string` as `strng` as the former is a "reserved word" in F#.

(b) Your JSON parser should read input from a file whose name is given on the command line. For example, suppose you have a file, `data.json`, which contains

```
[{"x":1,"y":2,"z":3},{"x":2,"y":3,"z":4}]
```

then you should be able to run your parser on this input like so

```
$ dotnet run data.json
```

(c) The output of your program should be a prettyprinted version of the input after parsing it. If the input is malformed, your program should print "Invalid JSON" and exit without printing anything else. Provide a function `prettyprint(j: JSON) : string` that turns an abstract syntax tree into a `string`. For example, when reading the previous `data.json` file, the program fragment

```
let filename = args[0]
let input = System.IO.File.ReadAllText filename
let ast_maybe = parse input
match ast_maybe with
| Some ast -> printfn "%s" (prettyprint ast)
| None     -> printfn "Invalid JSON"
```

should parse the file and print the JSON string back out. Do not worry about preserving indentation in your prettyprinted JSON.

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

i. Accept arbitrary amounts of whitespace between elements, like a real-world JSON parser. See the `examples/8-bonus.json` file for an example.

ii. Produce a `prettyprint` function that idents any nested element by four characters as in the example above. This will transform your program into a JSON formatter.

iii. Use the `diagnosticMessage` function to tell the user where their mistake is in the event that their input does not parse.

iv. Refer to the JSON specification (`https://datatracker.ietf.org/doc/html/rfc7159`), find any missing feature, and implement it in your parser.

If you decide to tackle any of the above, **be sure to tell me** that you attempted a bonus by including a file `BONUS.txt` that explains what you did. Otherwise I may not notice your hard work!

The last bonus item is challenging, but it is quite satisfying to produce a parser that can read in arbitrary JSON you can find on the internet. If you're looking to push yourself, give it a try! HINT: start with the empty array.

Tips:

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. Here are some tips you should follow to make your life easier.

(a) Start small and test often. The `examples` directory has sample inputs, ranked from easiest to hardest. Choose the easiest input first, and implement *only* the parsing function(s) that make that input work. Once you have satisfied yourself that your program works by testing it, choose the next easiest input and repeat.

(b) If you create your own test inputs, watch out where you put whitespace characters. Unless you explicitly write your parser to handle whitespace, it will not handle whitespace. Especially watch out for trailing newlines. Some editors (e.g., `nano`) like to insert them automatically. By default, the Visual Studio Code editor will not insert trailing newlines.

(c) 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
```

Place your files in your project root directory. You should be able to run your program on the command line by typing, for example, "`dotnet run data.json`" 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.