Homework 8

Due Sunday, April 24 by 10:00pm

Handout 21
CSCI 334: Spring 2022

Turn-In Instructions

Turn in your work using the Github partner repository assigned to you. The name of the Github repository will have the form `cs334lab8_<your user name>-<your partner's user name>`. For example, my repository might be `cs334lab8_dbarowy-wjannen`. You should have received an invite to commit in the repository in your email. If you did not receive this email, please contact me right away!

For this assignment, your completed submission should consist of two parts.

1. A LaTeX source file called `project.tex` and pre-built PDF called `project.pdf` in a folder called `project` that contains project-related writing. A LaTeX template is provided to help you get started.
2. For each coding 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`". 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.

This assignment is due on Sunday, April 24 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.

Honor Code

This is a partner project lab. You may work with another classmate if you wish, and you may co-develop solutions. Unlike with previous partner labs, you may share source code, and you only need to submit it once. In other words, there is no need to type up and submit your solution twice. As I already know who your partner is, you do not need to submit a `collaborators.txt` file to your repository. However, if your partner arrangement changes, please let me know.

Reading

1. **(Required)** "Parser Combinators"

Problems

Q1. (40 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 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# does not like us to use functions before we define them, which means that F# gets unhappy when it notices that we define parsers 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 *)
```

(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:

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.

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!

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. (0 points) ..... Final Project Presentation

What format do you prefer for the final project presentation? See `https://bit.ly/3KX4env` for a form with options.

Q3. (60 points) ..... Project Proposal

For this question, you will propose your final project: a programming language of your own design. You are strongly encouraged to work with a partner on this assignment, however you will be permitted to work by yourself if you feel up to the extra challenge.

At this stage, your proposal is still non-binding, however, you should proceed as if this is the language that you want to implement as your final project.

Many programming language specifications begin as informal proposals, and this is the template that we will follow in this class. With each stage of your project, you will revisit your document, making it clearer and more precise as you work. It will be a "living document." By the end of this class, your final specification should 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.

Structure of the Proposal

Version 1.0 of your specification should include the following sections. Please explicitly include these sections. The purpose of this document is to convince yourself that such a 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? Why does this problem need its own programming language?

(b) Design Principles

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

(c) Examples

3+ examples. Keeping in mind that your syntax is 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

As much as is needed. Sketch out the syntax of the language. For now, this can be an English description of the key syntactical elements and how they fit together. Examples are fine. We will eventually transform this into a formal syntax section written in Backus-Naur Form (BNF). If you prefer to cut to the chase and provide BNF, you are welcome to do so.

(f) Semantics

5+ paragraphs. 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, one per paragraph.

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

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

iii. How is your program represented? In other words, what components (types) will be used in your AST? If it helps you to think about this using ML algebraic data types, please use them. Otherwise, a rough sketch like a class hierarchy drawings or even a Java class file is OK.

iv. How do AST elements "fit together" to represent programs as abstract syntax? For the three example programs you gave earlier, provide sample abstract syntax trees.

v. 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?

   C. Evaluation is usually conceived of as a post-order traversal of an AST. Describe how such a traversal yields the effect you just described and *provide illustrations for clarity.* Demonstrate evaluation for at least one of your example programs.

Goals

Your language need not be (and I discourage you from trying to build) a Turing-complete programming language. Instead, focus on solving a small class of problems. In other words, design a *domain-specific programming language.*

Your project must eventually achieve all of the following objectives:

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

(b) It should have a parser that recognizes a grammatically-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 useful 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.

Sample Specifications

The following documents, described above in the readings section, are useful to see how others structure their language specifications. Skim these as necessary in order to get a feel for your proposal. Note that Standard ML is a large, extensively-developed general-purpose language and therefore has long and very detailed specification. Logo has short and imprecise specification as it is a special-purpose language.

Your specification should be as long as is necessary and no longer. Writing good technical documentation is a real art, and we can only scratch the surface in this class. In general, try to be concise and precise, but not so concise or precise that you sacrifice understandability. The Logo documentation, especially the section titled "A Logo Primer" is an example of lucid, but informal, technical writing that I think captures the ideas of the language beautifully, and without any formal specification.

How to Organize

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