98-008 Homework 1: PPM Parsing

Cooper Pierce cppierce@andrew.cmu.edu
Jack Duvall jrduvall@andrew.cmu.edu

Spring 2023

Overview

The goal of assignment is to get you used to basic Rust constructs and syntax, filling out the core of a program that actually does something useful! We hope this will give you familiarity with Rust's development workflow and start to get you comfortable with using the language.

This assignment will have you write a basic parser for a simple image format known as PPM.

PPM Images

PPM is a very simple image format, consisting of just:

- A "magic number" to distinguish it from other files
- The width and height of the image
- The maximum intensity value of
- Pixels as packed RGB pixel values, read sequentially row-by-row (i.e., row-major) from the top left of the image.

`http://ailab.eecs.wsu.edu/wise/P1/PPM.html` outlines the format; for simplicity, we'll summarize it again here. Note that we'll be using the binary format, where each pixel channel takes up exactly one byte.

PPM Grammar

The grammar for PPM files is as follows:

```
<ppm>        ::= <magic-num>
                 (<comment> | <whitespace>)+ <width>
                 (<comment> | <whitespace>)+ <height>
                 (<comment> | <whitespace>)+ <maxval>
                 \n <pixels>
<magic-num>  ::= P6\n
<whitespace> ::= \t | \n | \x0C | \r | ' ' (i.e., a literal space)
<comment>    ::= # [^\n]* \n
<width>      ::= [0-9]+
<height>     ::= [0-9]+
<maxval>     ::= [0-9]+
<pixels>     ::= [\x00-\xFF]*
```

There is also an additional constraint that the number of bytes parsed for pixels shall be exactly $3wh$ where w and h are the parsed width and height, respectively.

Some assistance with reading the table above, if you haven't seen BNF-like grammar specifications before:

- nonterminals (a name for a class of actual character sequences) are surrounded by angle brackets and are defined by production rules given on the right of a `::=`.
- the `|` operator expresses that either the left, or the right side can be chosen (e.g., `<whitespace>` can match any single newline, tab, etc..).
- the `+` and `*` operators express that the object to the left can be repeated 1 or more times or 0 or more times, respectively.
- ranges in square brackets express choice between the characters comprising the range; or the complement, if the initial character is a caret.

PPM Image Example

An example would probably help understand the above specification a bit:

```
P6
# I'm a PPM file! You can tell by the magic header
3 3 # This first number is the width, and the second is the height
# This next number is the maximum intensity of each pixel
# Exporting as "raw" in GIMP always gives 255, which makes sense; this is the
# maximum value of a u8. After this maxval, we are only allowed a single newline
# before the pixels start. These pixels can be arbitrary bytes!
255
<<<<<<<<<aaaaaaaaa~~~~~~~~~
```

Copy-pasting this text into a file ending with `.ppm` and opening it in a compatible image-viewing program should give a 3x3 (note that this means it may be quite small) image with 3 horizontal gray stripes.

What You Will Write

You will be writing code that parses the metadata in the PPM header, up until the pixels.

Files

This comprises editing the function `parse`, defined in `src/ppm.rs`.**This is the only function you should edit**.

This function uses a state machine to parse the ppm file. We've set you up with a couple `enum`s which collectively represent the state of the machine to execute the parsing steps, as well as a description of what each state represents and what you'll need to do. Then, you can just edit the main loop in `parse` to do what the comments tell you.

The file `src/main.rs` is a harness that wraps your code and uses OpenGL to display the resulting parsed image. You can compare this with a standard utility to view images, like ImageMagick's `display`. **You do not have to edit this file**.

Testing

To test your code you can use the provided PPM files, as well as writing your own. We recommend printing out some smaller ones first, to ensure the parsed result is what you expect, but for the larger ones you will be able to visually compare. Additionally, we've incorporated some of the

testing features in rustc/Cargo so you can automatically test your code on the provided files. Running `cargo test` will run these tests, which you can see at the bottom of `src/main.rs`.

Submitting

Submission will be on Gradescope. If you aren't in gradescope, message us in Discord or via email at `rust-stuco-staff@lists.andrew.cmu.edu`.

You should submit a zip file containing the whole crate rooted at the directory where Cargo.toml appears (e.g., run `zip submission.zip Cargo.toml src/*`) and upload that.