HTMLGEN: A HTML templating engine

Gopiandcode

2018

Contents

1 HtmlGen

Does the following sound familiar?

> A: "All I want is to template some files for a static site."
> B: "Oh, okay, that's simply, just install npm, and packages yarn,gulp,react,express...".
> A: "..."

Introducing HTMLGEN, a simple little standalone static templating engine for HTML. Coded using literate programming.

2 Usage

The command line interface for the program is as follows.

```
Usage:
  htmlgen [OPTIONS] [BASEDIR]

Simple templating engine for html documents

Positional arguments:
  BASEDIR               The project directory. If not specified, then --input,
                        --template and --output flags must be given.

Optional arguments:
  -h,--help             Show this help message and exit
  -o,--output OUTPUT    Directory for the output files to be saved. It defaults
                        to BASEDIR/bin
  -t,--template TEMPLATE
                        Directory to be searched to find templates. It defaults
                        to BASEDIR/template
  -i,--input INPUT      Directory in which the source files to be compiled are
                        located. It defaults to BASEDIR/bin
  -e,--error ERROR      Fail on the first undefined parameter
  -d,--default DEFAULT  Additional mapping for storing default values. If not
                        specified, the environment variable GOP_HTML_DEFAULTS
                        if defined, is used as a default.
```

3 Preamble

While the overall nature of this project is quite simple - just a bit of file loading and exports, we can leverage rust's ecosystem to make our development a little easier.

3.1 Crates

The crates we'll be using are as follows:

- **ArgParse** - This is a crate that I used a while back when making another command line application. It provides a very nice rustic interface over a library which produces command line interfaces compliant with most unix/linux standards.

an old fashioned regex.

```
extern crate argparse;
```

- **Regex** - We'll be taking advantage of this regex crate to make the parsing phase a little easier; while the stdlib provides some pretty useful string matching utilities, they don't quite match up to

```
extern crate regex;
```

3.2 Standard Library Imports

We'll be using the path utilities provided by the standard library to help us navigate the filesystem in a cross platform way.

```
use std::env;
use std::path;
use std::fs::File;
use std::io::Read;
use std::path::Path;
use std::io::Write;
```

3.3 Module structure

We'll be splitting up our codebase as follows:

```
mod crawler;
mod parser;
mod generator;
```

4 Command Line Interface

Clearly this project is going to be a command line application, as the static generator will need to parse a document and construct the components.

Using argparse - as imported in the preamble, we'll design a sweet and sexy interface to access our application. The main actions we'll allow a user to perform using this application will be as follows:

- **specify output folder** - by default the output of the compiled files are placed in `./bin/` dir, which is made if it does not exist.

- **specify template folder** - within a non-templated file, when a template reference is used, by default the application searches the `./template/` dir to resolve these references.
- **specify input folder** - by default the program searches `./src/` for the source files to be compiled

```
fn main() {
 let mut output_path = String::from("");
 let mut template_path = String::from("");
 let mut input_path = String::from("");
 let mut base_dir : Option<String> = None;
 //
↪   [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20strategy][high
↪   level error strategy]]
 let mut opt_strat =
↪  generator::GeneratorErrorCoreStrategy::Fail;
 // high level error strategy ends here
 //
↪   [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20strategy][high
↪   level error strategy]]
 let mut def_strat = None;
 // high level error strategy ends here
 let mut help_string : Vec<u8> = Vec::new();
 {
     let mut ap = argparse::ArgumentParser::new();
     ap.set_description("Simple templating engine for html
↪  documents");
     ap.refer(&mut output_path)
     .add_option(&["-o","--output"],
                 argparse::Store,
                 "Directory for the output files to be saved.
↪  It defaults to BASEDIR/bin");

     ap.refer(&mut template_path)
     .add_option(&["-t","--template"],
                 argparse::Store,
                 "Directory to be searched to find templates.
↪  It defaults to BASEDIR/template");
```

```
    ap.refer(&mut input_path)
    .add_option(&["-i","--input"],
                argparse::Store,
                "Directory in which the source files to be
↪  compiled are located. It defaults to BASEDIR/bin");

    ap.refer(&mut base_dir)
    .add_argument("BASEDIR",
          argparse::StoreOption,
          "The project directory. If not specified, then
↪  --input, --template and --output flags must be given. ");

    //
↪  [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20args][high
↪  level error args]]
    ap.refer(&mut opt_strat)
      .add_option(&["-e", "--error"],
                  argparse::Store,
                  "Fail on the first undefined parameter");
    // high level error args ends here
    //
↪  [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20args][high
↪  level error args]]
    ap.refer(&mut def_strat)
      .add_option(&["-d", "--default"],
                  argparse::StoreOption,
                  "Additional mapping for storing default
↪  values. If not specified, the environment variable
↪  GOP_HTML_DEFAULTS if defined, is used as a default.");
    // high level error args ends here

    ap.print_help("htmlgen", &mut help_string);

    ap.parse_args_or_exit();
}
let help_string = unsafe {
↪  String::from_utf8_unchecked(help_string) };
let mut output_path = if output_path.is_empty() { None } else
↪  { Some(output_path) };
```

```
let mut template_path = if template_path.is_empty() { None }
↪  else { Some(template_path) };
let mut input_path = if input_path.is_empty() { None } else {
↪  Some(input_path) };
if base_dir.is_none() && (output_path.is_none() ||
↪  template_path.is_none() || input_path.is_none()) {
   println!("{}", help_string);
   ::std::process::exit(-1);
}
let (output_path, template_path, input_path) = if let Some(bd)
↪  = base_dir {
    let bd = Path::new(&bd);
    let error_string = format!("{:?} is not a valid path",
↪  bd);
    let alt_output_path =
↪  bd.join(Path::new(&"bin")).to_str().expect(&error_string).to_owned();
    let alt_template_path =
↪  bd.join(Path::new(&"template")).to_str().expect(&error_string).to_owned();
    let alt_input_path =
↪  bd.join(Path::new(&"src")).to_str().expect(&error_string).to_owned();

    let output_path = output_path.unwrap_or_else(||
↪  alt_output_path );
    let template_path = template_path.unwrap_or_else(||
↪  alt_template_path );
    let input_path = input_path.unwrap_or_else(||
↪  alt_input_path );

    (output_path, template_path, input_path)
} else {
    (output_path.unwrap(), template_path.unwrap(),
↪  input_path.unwrap())
};
let output_directory = Path::new(&output_path);
let input_directory = Path::new(&input_path);
let template_directory = Path::new(&template_path);
//
↪  [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20update][high
↪  level error update]]
```

```
 let def_strat = def_strat.or_else(||
↪  env::var("GOP_HTML_DEFAULTS").ok());
 // high level error update ends here
 //
↪  [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20update][high
↪  level error update]]
 let err_strat = match def_strat {
    None =>
       generator::GeneratorErrorStrategy::Base(opt_strat),
    Some(path) => {
       let mapping = {
           let def_path = Path::new(&path);
           if let Ok(mut file) = File::open(&def_path) {
              let mut def_source = String::new();
              if let Ok(_count) = file.read_to_string(&mut
↪  def_source) {
                  parser::parse_source_string(&def_source).ok()
              } else {
                  None
              }
           } else {
               None
           }
       };
       match mapping {
         Some((name, map)) =>
             generator::GeneratorErrorStrategy::Default(map,
↪  opt_strat),
         None => {
             eprintln!("Encountered error while reading default
↪  mapping at {:?}.", path);
             generator::GeneratorErrorStrategy::Base(opt_strat)
         }
       }
    }
 };
 // high level error update ends here
 println!("{:?}", crawler::crawl_directories(&output_directory,
↪  &input_directory, &template_directory, &err_strat));
}
```

We'll set up some initial variables to hold the parameters from the command line.

```
let mut output_path = String::from("");
let mut template_path = String::from("");
let mut input_path = String::from("");
let mut base_dir : Option<String> = None;
```

We'll also need to setup an error strategy - this will require some additional data structures, so we'll leave it to the end.

```
//
↪  [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20strategy][high
↪  level error strategy]]
let mut opt_strat =
↪  generator::GeneratorErrorCoreStrategy::Fail;
// high level error strategy ends here
//
↪  [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20strategy][high
↪  level error strategy]]
let mut def_strat = None;
// high level error strategy ends here
```

Using argparse, we can implement this cmdline interface as follows:

```
let mut help_string : Vec<u8> = Vec::new();
{
    let mut ap = argparse::ArgumentParser::new();
    ap.set_description("Simple templating engine for html
↪   documents");
    ap.refer(&mut output_path)
    .add_option(&["-o","--output"],
                argparse::Store,
                "Directory for the output files to be saved. It
↪   defaults to BASEDIR/bin");

    ap.refer(&mut template_path)
    .add_option(&["-t","--template"],
                argparse::Store,
                "Directory to be searched to find templates. It
↪   defaults to BASEDIR/template");
```

```
    ap.refer(&mut input_path)
    .add_option(&["-i","--input"],
                argparse::Store,
                "Directory in which the source files to be
↪   compiled are located. It defaults to BASEDIR/bin");

    ap.refer(&mut base_dir)
    .add_argument("BASEDIR",
          argparse::StoreOption,
          "The project directory. If not specified, then
↪   --input, --template and --output flags must be given. ");

    //
↪   [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20args][high
↪   level error args]]
    ap.refer(&mut opt_strat)
      .add_option(&["-e", "--error"],
                  argparse::Store,
                  "Fail on the first undefined parameter");
    // high level error args ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20args][high
↪   level error args]]
    ap.refer(&mut def_strat)
      .add_option(&["-d", "--default"],
                  argparse::StoreOption,
                  "Additional mapping for storing default
↪   values. If not specified, the environment variable
↪   GOP_HTML_DEFAULTS if defined, is used as a default.");
    // high level error args ends here

    ap.print_help("htmlgen", &mut help_string);

    ap.parse_args_or_exit();
}
```

Before we do anything, let's get a copy of the help string generated by `argparse` for the program.

```
let help_string = unsafe {
↪   String::from_utf8_unchecked(help_string) };
```

Additionally, we'll convert the unwritten values to options.

```
let mut output_path = if output_path.is_empty() { None } else {
↪  Some(output_path) };
let mut template_path = if template_path.is_empty() { None }
↪  else { Some(template_path) };
let mut input_path = if input_path.is_empty() { None } else {
↪  Some(input_path) };
```

Following this, we do some error checking to ensure that everything is suitably specified. If the base directory is not specified, then all other parameters must be specified - otherwise we exit.

```
if base_dir.is_none() && (output_path.is_none() ||
↪  template_path.is_none() || input_path.is_none()) {
   println!("{}", help_string);
   ::std::process::exit(-1);
}
```

With that done, we can safely extract the paths. As specified, our output and template paths take default values from the supplied `BASEDIR`.

```
let (output_path, template_path, input_path) = if let Some(bd)
↪  = base_dir {
    let bd = Path::new(&bd);
    let error_string = format!("{:?} is not a valid path", bd);
    let alt_output_path =
↪  bd.join(Path::new(&"bin")).to_str().expect(&error_string).to_owned();
    let alt_template_path =
↪  bd.join(Path::new(&"template")).to_str().expect(&error_string).to_owned();
    let alt_input_path =
↪  bd.join(Path::new(&"src")).to_str().expect(&error_string).to_owned();

    let output_path = output_path.unwrap_or_else(||
↪  alt_output_path );
    let template_path = template_path.unwrap_or_else(||
↪  alt_template_path );
    let input_path = input_path.unwrap_or_else(||
↪  alt_input_path );

    (output_path, template_path, input_path)
} else {
    (output_path.unwrap(), template_path.unwrap(),
↪   input_path.unwrap())
};
```

5 Core Logic

Now we've obtained the directory for the files to be stored, we can move on to the main logic of the program. Fundamentaly the logic of this program can be split into two main components:

- Recursively descending the source directory, keeping track of the file structure.

```
mod crawler;
```

- Extracting the data from a given file

```
mod parser;
```

- generate a compiled html file from the template and save it to a folder

```
mod generator;

let output_directory = Path::new(&output_path);
let input_directory = Path::new(&input_path);
let template_directory = Path::new(&template_path);
```

Thus the high level execution of the system is as follows. First we update the error strategy.

```
//
↪   [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20update][high
↪   level error update]]
let def_strat = def_strat.or_else(||
↪  env::var("GOP_HTML_DEFAULTS").ok());
// high level error update ends here
//
↪   [[file:~/Documents/html_gen/html_gen.org::high%20level%20error%20update][high
↪   level error update]]
let err_strat = match def_strat {
   None =>
```

```
        generator::GeneratorErrorStrategy::Base(opt_strat),
    Some(path) => {
        let mapping = {
            let def_path = Path::new(&path);
            if let Ok(mut file) = File::open(&def_path) {
                let mut def_source = String::new();
                if let Ok(_count) = file.read_to_string(&mut
↪  def_source) {
                    parser::parse_source_string(&def_source).ok()
                } else {
                    None
                }
            } else {
                None
            }
        };
        match mapping {
          Some((name, map)) =>
              generator::GeneratorErrorStrategy::Default(map,
↪  opt_strat),
          None => {
              eprintln!("Encountered error while reading default
↪  mapping at {:?}.", path);
              generator::GeneratorErrorStrategy::Base(opt_strat)
          }
        }
    }
};
// high level error update ends here
```

Then we run the crawler and print the output. Done.

```
println!("{:?}", crawler::crawl_directories(&output_directory,
↪  &input_directory, &template_directory, &err_strat));
```

5.1 Parser Logic

Before we begin, we'll need the following packages in our parser:

```
use std::collections::HashMap;
use regex::Regex;
```

```
#[derive(Debug)]
pub enum ParseError {
   TemplateNotFound,
   InvalidIdentifier,
   UnterminatedBody
}
```

Once again, our core specification for the parser is to extract a set of key value pairs. Our syntax will be of the following form:

```
ID := (Sigma/{:, (, )})+
INTRO := #+template: Sigma+\n
MAPPING := ID:  ((SIGMA/{¬})|\¬)* ¬
DOCUMENT := INTRO MAPPING*
```

Our parser will take in a string (the contents of the file), and return either a hashmap of values and a template name, or an error.

```
fn split_at_regex<'a>(string: &'a str, pat: &Regex) -> (&'a
↪  str, &'a str) {
  if let Some(m) = pat.find(string) {
     string.split_at(m.end())
  } else {
     (&"", string)
  }
}
fn split_at_pattern<'a>(string: &'a str, pat: &str) -> (&'a
↪  str, &'a str) {
  if let Some(ind) = string.find(pat) {
     string.split_at(ind)
  } else {
     (&"", string)
  }
}

pub fn parse_source_string(source: &str)
   -> Result<(String, HashMap<String,String>),ParseError> {
let key_regex = Regex::new("^[^¬:{}\\\\]*:").unwrap();
let data_regex =
↪  Regex::new("^(\\\\¬|([^¬\\\\]|\\\\[^¬])*)*¬").unwrap();
if !source.trim_left().starts_with("#+template:") {
   return Err(ParseError::TemplateNotFound);
}
let source = source.trim_left().split_at(11).1;
let (raw_template_name, remaining_string) =
↪  split_at_pattern(source, "\n");
let template_name = raw_template_name.trim();
if template_name.is_empty() {
   return Err(ParseError::TemplateNotFound);
}
let mut data : HashMap<String, String> = HashMap::new();
let mut completed = false;
let mut source = remaining_string;
let mut data = data;

while !completed {
   //
↪  [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪  pairs loop]]
   let (raw_key_name, remaining_string) =
↪  split_at_regex(source, &key_regex);
   let key_name = raw_key_name.trim();
   source = remaining_string;
   // source pairs loop ends here
   //
↪  [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪  pairs loop]]
   if key_name.len() == 0 {
     eprintln!("Invalid parse, found empty/malformed ID tag");
     return Err(ParseError::InvalidIdentifier);
   }
   // source pairs loop ends here
   //
↪  [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪  pairs loop]]
   let mut key_name = key_name.to_string();
   key_name.pop();
   let key_name = key_name.trim();
   // source pairs loop ends here
```

```
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    let (raw_data, remaining_string) = split_at_regex(source,
↪   &data_regex);
    let src_data = raw_data.trim();
    source = remaining_string;
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    if src_data.len() == 0 {
      eprintln!("Invalid parse, found body with no terminating
↪   tag.");
      return Err(ParseError::UnterminatedBody);
    }
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    let mut src_data = src_data.to_string();
    src_data.pop();
    let src_data = src_data.trim();
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    data.insert(key_name.to_string(), src_data.to_string());
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    if source.trim().is_empty() {
        break;
    }
    // source pairs loop ends here
}
Ok((template_name.to_string(), data))
}
```

```rust
#[cfg(test)]
mod test {
   use super::*;

  #[test]
  fn must_start_with_template_directive() {
     assert!(parse_source_string("temp-justkidding\n id:\n
↪  #+template:\n").is_err());
  }
  #[test]
  fn must_provide_template_name() {
      assert!(parse_source_string("#+template:
↪  example\n").is_ok());
      assert!(parse_source_string("#+template:\n").is_err());
      assert!(parse_source_string("#+template:
↪  \n").is_err());
      assert!(parse_source_string("#+template:   \n
↪  \n").is_err());
      assert!(parse_source_string("#+template:   \t
↪  \n").is_err());
  }
}
```

Where a parsing error will be one of the following:

- **Template not found** - if the source file does not specify a template to be loaded
- **Invalid identifier** - if an identifier contains an invalid character.
- **Unterminated Body** - if a body does not have a valid terminator.

```rust
#[derive(Debug)]
pub enum ParseError {
   TemplateNotFound,
   InvalidIdentifier,
   UnterminatedBody
}
```

For simplicity, we're making the parser as general as possible and opting to make failure as unlikely as possible.

To do the parsing, first we start off by consuming the template directive, and failing if not present.

First, we check that the template contains a template directive - we're leaving resolving the template to a file to a later point.

```
if !source.trim_left().starts_with("#+template:") {
   return Err(ParseError::TemplateNotFound);
}
```

This means that if a source does not start with a directive, its parsing will fail:

```
#[test]
fn must_start_with_template_directive() {
   assert!(parse_source_string("temp-justkidding\n id:\n
↪  #+template:\n").is_err());
}
```

After this check, we can safely consume the first part of the string.

```
let source = source.trim_left().split_at(11).1;
```

Next, let's retrieve the actual template name - failing if it was not provided.

```
let (raw_template_name, remaining_string) =
↪  split_at_pattern(source, "\n");
let template_name = raw_template_name.trim();
if template_name.is_empty() {
   return Err(ParseError::TemplateNotFound);
}
```

This also means that if a source does not provide a template name its parsing will fail:

```
#[test]
fn must_provide_template_name() {
    assert!(parse_source_string("#+template:
↪  example\n").is_ok());
    assert!(parse_source_string("#+template:\n").is_err());
    assert!(parse_source_string("#+template:    \n").is_err());
    assert!(parse_source_string("#+template:   \n
↪  \n").is_err());
```

```
    assert!(parse_source_string("#+template:   \t
↪  \n").is_err());
}
```

Now, our remaining task is to simply iterate through the remaining `ID: DATA` pairs, and accumulate these values into a hashmap - let's begin by setting up an initial hashmap to store the files.

```
let mut data : HashMap<String, String> = HashMap::new();
```

Next, we'll define a simple loop to do the accumulation - it will use a reference to the hashmap, and the source:

```
let mut completed = false;
let mut source = remaining_string;
let mut data = data;

while !completed {
   //
↪  [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪  pairs loop]]
   let (raw_key_name, remaining_string) =
↪  split_at_regex(source, &key_regex);
   let key_name = raw_key_name.trim();
   source = remaining_string;
   // source pairs loop ends here
   //
↪  [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪  pairs loop]]
   if key_name.len() == 0 {
     eprintln!("Invalid parse, found empty/malformed ID tag");
     return Err(ParseError::InvalidIdentifier);
   }
   // source pairs loop ends here
   //
↪  [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪  pairs loop]]
   let mut key_name = key_name.to_string();
   key_name.pop();
   let key_name = key_name.trim();
   // source pairs loop ends here
```

```
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    let (raw_data, remaining_string) = split_at_regex(source,
↪   &data_regex);
    let src_data = raw_data.trim();
    source = remaining_string;
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    if src_data.len() == 0 {
      eprintln!("Invalid parse, found body with no terminating
↪   tag.");
      return Err(ParseError::UnterminatedBody);
    }
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    let mut src_data = src_data.to_string();
    src_data.pop();
    let src_data = src_data.trim();
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    data.insert(key_name.to_string(), src_data.to_string());
    // source pairs loop ends here
    //
↪   [[file:~/Documents/html_gen/html_gen.org::source%20pairs%20loop][source
↪   pairs loop]]
    if source.trim().is_empty() {
        break;
    }
    // source pairs loop ends here
}
```

To extract the keys and bodies, we'll be using a regex - it checks that the start of the string consists of non terminator characters, followed by a colon.

```
let key_regex = Regex::new("^[^¬:{}\\\\]*:").unwrap();
```

Now, inside the loop, we'll use the regex to extract the key values - for this purpose, we'll define a custom `split_by_regex` function, which operates like the `split_at_pattern` function but uses the first match of a regex to split the input.

```
fn split_at_regex<'a>(string: &'a str, pat: &Regex) -> (&'a
 ↪  str, &'a str) {
  if let Some(m) = pat.find(string) {
     string.split_at(m.end())
  } else {
     (&"", string)
  }
}
```

Now, using this function, we can implement the key extraction.

```
let (raw_key_name, remaining_string) = split_at_regex(source,
 ↪  &key_regex);
let key_name = raw_key_name.trim();
source = remaining_string;
```

Now due to the way we're extracting the values, bad input may lead to an incorrect parse - we'll try and avoid this by printing an error when the IDs are wrong:

```
if key_name.len() == 0 {
  eprintln!("Invalid parse, found empty/malformed ID tag");
  return Err(ParseError::InvalidIdentifier);
}
```

Due to the way we extract the ids, we also end up bringing the colon as well. Let's just remove it before proceeding:

```
let mut key_name = key_name.to_string();
key_name.pop();
let key_name = key_name.trim();
```

Now we can move on to extracting the data. Let's start by defining a regular expression to isolate specific syntax we wish to capture.

```
let data_regex =
 ↪  Regex::new("^(\\\\¬|([^¬\\\\]|\\\\[^¬])*)*¬").unwrap();
```

The regex we're using can be explained as follows; the outermost kleene closure captures the main constraint that the data should start from the start of the string and end at the first occurrance of a terminating character.

```
^ INTERNALS *¬
```

Next, for the contents of a body, we have to capture 2 main cases:

- When the character is normal and non interesting
- When the character is an escaped terminator.

```
INTERNALS ::= (ESCAPED_TERMINATOR|NORMAL_CHARACTERS)
```

For the escaped terminator case, we simply match on a backspace followed by a terminator.

```
ESCAPED_TERMINATOR = \¬
```

In the case of normal characters, either

- the character is neither a backslash or a terminator
- the character is a backslash and is followed by anything other than a terminator

```
NORMAL_CHARACTERS = ([^¬\\\\]|\\\\[^¬])*
```

Using this regex we can trivially extract the data, repeating the code for key extraction.

```
let (raw_data, remaining_string) = split_at_regex(source,
↪  &data_regex);
let src_data = raw_data.trim();
source = remaining_string;
```

While it is fine for data to be empty, we always require the user to provide the end character, so the string should never be 0.

```
if src_data.len() == 0 {
  eprintln!("Invalid parse, found body with no terminating
↪  tag.");
  return Err(ParseError::UnterminatedBody);
}
```

Now, as before, let's remove the terminating character.

```
let mut src_data = src_data.to_string();
src_data.pop();
let src_data = src_data.trim();
```

Finally, now we've extracted the id and the tag, we can simply put the values into our hashmap.

```
data.insert(key_name.to_string(), src_data.to_string());
```

Now, we also need to check for a terminating condition - we'll do this by checking if the remaining string, when trimmed, is empty.

```
if source.trim().is_empty() {
    break;
}
```

Finally, now that string has been consumed, we can simply return the template name and the populated hashmap.

```
Ok((template_name.to_string(), data))
```

Aside: Notice, that during the parsing, we're using our own custom function to allow us to split by a pattern, a feature the stdlib doesn't seem to provide.

This utility function splits a string by the first occurance of a pattern returning a string up to the first occurrance of the pattern and a string continuing from the pattern - the second string contains the text matching the pattern.

```
fn split_at_pattern<'a>(string: &'a str, pat: &str) -> (&'a
↪  str, &'a str) {
  if let Some(ind) = string.find(pat) {
     string.split_at(ind)
  } else {
     (&"", string)
  }
}
```

5.2 Generator Logic

The generator takes in an input templated string and an associated mapping and returns a string in which the templates have been filled - it also takes in a paramter dictating how to respond to ill formed strings.

We'll be importing the following libraries to make this thing work.

```
use std::collections::HashMap;
use regex::{Regex, Captures};
```

The generator module follows the standard pattern.

```
use std::collections::HashMap;
use regex::{Regex, Captures};
use std::str::FromStr;
#[derive(Clone,Debug,PartialEq)]
pub enum GeneratorErrorCoreStrategy {
   Fail,
   Ignore,
   Fixed(String)
}
pub enum GeneratorErrorStrategy {
   Base(GeneratorErrorCoreStrategy),
   Default(HashMap<String,String>, GeneratorErrorCoreStrategy)
}
#[derive(Debug)]
pub enum GeneratorError {
  UndefinedParameter
}
impl FromStr for GeneratorErrorCoreStrategy {
    type Err = ();
    fn from_str(src: &str) ->
↪   Result<GeneratorErrorCoreStrategy, ()> {
        return match src {
            "fail" => Ok(GeneratorErrorCoreStrategy::Fail),
            "ignore" => Ok(GeneratorErrorCoreStrategy::Ignore),
            x => {
                if let Some(ind) = src.find("=") {
                   if ind + 1 < src.len() {
                       let (txt, rem) = src.split_at(ind+1);
                       if txt == "fixed=" {
```

```
 ↪  Ok(GeneratorErrorCoreStrategy::Fixed(rem.to_string()))
                        } else {
                            Err(())
                        }
                    } else {
                        Err(())
                    }
                 } else {
                   Err(())
                 }
            },
        };
    }
}

pub fn generate_output(input: String, mapping: HashMap<String,
 ↪  String>, fail_response: &GeneratorErrorStrategy) ->
 ↪  Result<String, GeneratorError> {
 let parameter_regex =
 ↪  Regex::new("\\{([^¬:{}\\\\]*)\\}").unwrap();
 let mut lookup_failed = false;
 let new_string = parameter_regex.replace_all(&input, |caps:
 ↪  &Captures| {
    if let Some(value) = mapping.get(&caps[1]) {
       value
    } else {
       match &fail_response {
           GeneratorErrorStrategy::Base(strategy) => {
               match strategy {
                 GeneratorErrorCoreStrategy::Fail => {
                     lookup_failed = true;
                     ""
                 }
                 GeneratorErrorCoreStrategy::Ignore => {
                     &caps[0]
                 },
                 GeneratorErrorCoreStrategy::Fixed(text) => {
                     text
                 }
```

```
                }
            }
            GeneratorErrorStrategy::Default(mapping, strategy)
↪   => {
                if let Some(value) = mapping.get(&caps[1]) {
                    value
                } else {
                    match strategy {
                        GeneratorErrorCoreStrategy::Fail => {
                            lookup_failed = true;
                            ""
                        }
                        GeneratorErrorCoreStrategy::Ignore => {
                            &caps[0]
                        },
                        GeneratorErrorCoreStrategy::Fixed(text) =>
↪   {
                            text
                        }
                    }
                }
            }
        }
    }
 });
 if lookup_failed {
    return Err(GeneratorError::UndefinedParameter);
 }
 Ok(new_string.to_string())
}

#[cfg(test)]
mod tests {
   use super::*;

   #[test]
   fn from_st_works() {
      assert_eq!(GeneratorErrorCoreStrategy::from_str("ignore"),
↪   Ok(GeneratorErrorCoreStrategy::Ignore));
```

```
     assert_eq!(GeneratorErrorCoreStrategy::from_str("fail"),
↪   Ok(GeneratorErrorCoreStrategy::Fail));

↪   assert_eq!(GeneratorErrorCoreStrategy::from_str("fixed=missing"),
↪   Ok(GeneratorErrorCoreStrategy::Fixed("missing".to_string())));
   }
}
```

The main utility provided by the generator is the main function that populates the templated string when given a mapping, additionally we must specify how the generator should respond when missing templates are found.

```
pub fn generate_output(input: String, mapping: HashMap<String,
↪   String>, fail_response: &GeneratorErrorStrategy) ->
↪   Result<String, GeneratorError> {
 let parameter_regex =
↪   Regex::new("\\{([^¬:{}\\\\]*)\\}").unwrap();
 let mut lookup_failed = false;
 let new_string = parameter_regex.replace_all(&input, |caps:
↪   &Captures| {
    if let Some(value) = mapping.get(&caps[1]) {
       value
    } else {
       match &fail_response {
           GeneratorErrorStrategy::Base(strategy) => {
               match strategy {
                 GeneratorErrorCoreStrategy::Fail => {
                     lookup_failed = true;
                     ""
                 }
                 GeneratorErrorCoreStrategy::Ignore => {
                     &caps[0]
                 },
                 GeneratorErrorCoreStrategy::Fixed(text) => {
                     text
                 }
               }
           }
           GeneratorErrorStrategy::Default(mapping, strategy)
↪   => {
               if let Some(value) = mapping.get(&caps[1]) {
                  value
               } else {
                  match strategy {
                    GeneratorErrorCoreStrategy::Fail => {
                        lookup_failed = true;
                        ""
                    }
                    GeneratorErrorCoreStrategy::Ignore => {
                        &caps[0]
                    },
                    GeneratorErrorCoreStrategy::Fixed(text) =>
↪  {
                        text
                    }
                  }
               }
           }
       }
    }
 });
 if lookup_failed {
    return Err(GeneratorError::UndefinedParameter);
 }
 Ok(new_string.to_string())
}
```

The strategies the generator should accept are:

- **Fail** - Error out if a parameter that is not in the mapping is found in the template; this is the default.
- **Ignore** - ignore any missing parameters.
- **Fixed** - replace any missing parameters with a fixed response
- **Default** - try a default mapping for the keyword, otherwise try one of the other strategies.

To implement this, we'll use two structures, one to represent the non-recursive cases, and the other for the default option.

```
#[derive(Clone,Debug,PartialEq)]
pub enum GeneratorErrorCoreStrategy {
   Fail,
   Ignore,
   Fixed(String)
}
```

Thus for the full enum, we can avoid having to mess with boxes.

```
pub enum GeneratorErrorStrategy {
    Base(GeneratorErrorCoreStrategy),
    Default(HashMap<String,String>, GeneratorErrorCoreStrategy)
}
```

Now, the errors the templating function can return are partially based on the error response strategies.

- **Undefined Parameter** - An error when a paremeter with no mapping is found, and the strategy is sufficiently strict.

```
#[derive(Debug)]
pub enum GeneratorError {
  UndefinedParameter
}
```

The core logic of the generator is to use capture groups capabilities provided by the regex crate.

We'll reuse the same pattern as used in the parser, but wrap it in braces and capture the contents.

```
let parameter_regex =
↪  Regex::new("\\{([^¬:{}\\\\]*)\\}").unwrap();
```

Before we run the regex, we'll need to set up some variables to capture lookup errors.

```
let mut lookup_failed = false;
```

Next, we'll run the regex on the input string.

```
let new_string = parameter_regex.replace_all(&input, |caps:
↪  &Captures| {
    if let Some(value) = mapping.get(&caps[1]) {
        value
    } else {
        match &fail_response {
            GeneratorErrorStrategy::Base(strategy) => {
                match strategy {
                  GeneratorErrorCoreStrategy::Fail => {
                      lookup_failed = true;
                      ""
                  }
                  GeneratorErrorCoreStrategy::Ignore => {
                      &caps[0]
                  },
                  GeneratorErrorCoreStrategy::Fixed(text) => {
                      text
                  }
                }
            }
            GeneratorErrorStrategy::Default(mapping, strategy) =>
↪  {
                if let Some(value) = mapping.get(&caps[1]) {
                   value
                } else {
                   match strategy {
                     GeneratorErrorCoreStrategy::Fail => {
                         lookup_failed = true;
                         ""
                     }
                     GeneratorErrorCoreStrategy::Ignore => {
                         &caps[0]
                     },
                     GeneratorErrorCoreStrategy::Fixed(text) => {
                         text
                     }
                   }
                }
            }
        }
   }
});
```

If a lookup failed, then we'll return an error.

```
if lookup_failed {
   return Err(GeneratorError::UndefinedParameter);
}
```

Once that's done we have the result string - it's a `Cow<str>` though, so we just need to do a conversion before returning it.

```
Ok(new_string.to_string())
```

All that's left is to define the replacement logic - if it matches, we can simply return the value stored in the hashmap.

```
if let Some(value) = mapping.get(&caps[1]) {
   value
} else {
   match &fail_response {
       GeneratorErrorStrategy::Base(strategy) => {
           match strategy {
             GeneratorErrorCoreStrategy::Fail => {
                 lookup_failed = true;
                 ""
             }
             GeneratorErrorCoreStrategy::Ignore => {
                 &caps[0]
             },
             GeneratorErrorCoreStrategy::Fixed(text) => {
                 text
             }
           }
       }
       GeneratorErrorStrategy::Default(mapping, strategy) => {
           if let Some(value) = mapping.get(&caps[1]) {
              value
           } else {
              match strategy {
                GeneratorErrorCoreStrategy::Fail => {
                    lookup_failed = true;
                    ""
                }
                GeneratorErrorCoreStrategy::Ignore => {
                    &caps[0]
                },
                GeneratorErrorCoreStrategy::Fixed(text) => {
                    text
                }
              }
           }
       }
   }
}
```

If the lookup failes, our action depends on the error strategy we've chosen.

```
match &fail_response {
    GeneratorErrorStrategy::Base(strategy) => {
        match strategy {
          GeneratorErrorCoreStrategy::Fail => {
              lookup_failed = true;
              ""
          }
          GeneratorErrorCoreStrategy::Ignore => {
              &caps[0]
          },
          GeneratorErrorCoreStrategy::Fixed(text) => {
              text
          }
        }
    }
    GeneratorErrorStrategy::Default(mapping, strategy) => {
        if let Some(value) = mapping.get(&caps[1]) {
           value
        } else {
           match strategy {
             GeneratorErrorCoreStrategy::Fail => {
                 lookup_failed = true;
                 ""
             }
             GeneratorErrorCoreStrategy::Ignore => {
                 &caps[0]
             },
             GeneratorErrorCoreStrategy::Fixed(text) => {
                 text
             }
           }
        }
    }
}
```

For the base case, we simply match on the specific strategy chosen to decide our action.

```
match strategy {
  GeneratorErrorCoreStrategy::Fail => {
      lookup_failed = true;
      ""
  }
  GeneratorErrorCoreStrategy::Ignore => {
      &caps[0]
  },
  GeneratorErrorCoreStrategy::Fixed(text) => {
      text
  }
}
```

If the strategy is a fail fast case, then we still return an empty string, but we set the lookup failed error, thereby ensuring that the result of the call is an error.

```
lookup_failed = true;
""
```

If the strategy is an ignore case, we simply leave the parameter as it was.

```
&caps[0]
```

For the fixed case, we just return the fixed string.

```
text
```

Now, for the default mapping case, we first check if the default mapping contains a value for the parameter. If it does, we can simply return that value.

```
if let Some(value) = mapping.get(&caps[1]) {
   value
} else {
   match strategy {
     GeneratorErrorCoreStrategy::Fail => {
         lookup_failed = true;
         ""
     }
     GeneratorErrorCoreStrategy::Ignore => {
         &caps[0]
     },
     GeneratorErrorCoreStrategy::Fixed(text) => {
         text
     }
   }
}
```

If it doesn't, we simply match on the error strategy as previous.

```
match strategy {
  GeneratorErrorCoreStrategy::Fail => {
      lookup_failed = true;
      ""
  }
  GeneratorErrorCoreStrategy::Ignore => {
      &caps[0]
  },
  GeneratorErrorCoreStrategy::Fixed(text) => {
      text
  }
}
```

5.3 Crawler Logic

The core logic for the crawler is to descend the input directory, keeping track of the current path, pass each file through the parser, then pass on the generated mapping to the generator, along with a corresponding template file and output file.

We'll be importing the following libraries for doing the core logic.

```
use std::fs;
use std::io::Read;
use std::fs::File;
use std::path::Path;
use std::convert::AsRef;
```

We'll also be bringing in the parsing function from the parser, and the generator function from the generator.

```
use parser::{parse_source_string,ParseError};
use generator::{generate_output, GeneratorError,
 ↪  GeneratorErrorStrategy};
```

The main structure for the crawler is as follows.

```
use std::fs;
use std::io::Read;
use std::fs::File;
use std::path::Path;
use std::convert::AsRef;
use parser::{parse_source_string,ParseError};
use generator::{generate_output, GeneratorError,
 ↪  GeneratorErrorStrategy};

#[derive(Debug)]
pub enum CrawlError {
  ParseError(ParseError),
  GeneratorError(GeneratorError),
  TemplateNotFound(String),
  InputDirectoryError,
  OutputFileError(String),
  InputFileError(String),
}

pub fn crawl_directories<P,Q,R>(
    output_directory: &P,
    input_directory: &Q,
    template_path: &R,
    err_strat: &GeneratorErrorStrategy
) -> Result<u32,CrawlError>
 where P : AsRef<Path>,
       Q : AsRef<Path>,
       R : AsRef<Path> {
let mut file_count = 0;
let input_files = input_directory.as_ref()
                  .read_dir()
                  .map_err(|_|
                        CrawlError::InputDirectoryError
                  )?;
for input_file in input_files {
   let input_file = input_file.map_err(|e|
 ↪ CrawlError::InputFileError(format!("{:?}", e)))?;
   let input_metadata = input_file.metadata().map_err(|e|
 ↪ CrawlError::InputFileError(format!("{:?}", e)))?;
   let input_file_name = input_file.file_name();
   let input_file_path = input_file.path();
   let input_file_extension =
 ↪ input_file_path.extension().and_then(|ext| ext.to_str());
   let input_file_base =
 ↪ input_file_path.file_stem().and_then(|stem| stem.to_str());
   if input_metadata.is_dir() {
       let dir_name = Path::new(&input_file_name);
       let new_output_dir = output_directory
                            .as_ref()
                            .join(&dir_name);
       let new_input_dir = input_directory
                           .as_ref()
                           .join(&dir_name);
       let n_count = crawl_directories(
           &new_output_dir,
           &new_input_dir,
           template_path,
           err_strat
       )?;
       file_count += n_count;
   } else if input_metadata.is_file() && (input_file_extension
 ↪ == Some("gop")) && (input_file_base.is_some()) {
       let input_text = {
          let mut temp = String::new();
          let mut file =
    ↪ File::open(input_file.path()).map_err(|e|
    ↪ CrawlError::InputFileError(format!("{:?}", e)))?;
          file.read_to_string(&mut temp).map_err(|e|
 ↪ CrawlError::InputFileError(format!("{:?}", e)))?;
          temp
       };
       let (template_name, mapping) =
    ↪ parse_source_string(&input_text).map_err(|e|
    ↪ CrawlError::ParseError(e))?;
```

```
        let template_path =
    ↪   template_path.as_ref().join(&Path::new(&template_name));
        let template_text = {
            let mut temp = String::new();
            let mut file = File::open(template_path).map_err(|e|
    ↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
            file.read_to_string(&mut temp).map_err(|e|
    ↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
            temp
        };
        let result = generate_output(
            template_text,
            mapping,
            err_strat
        ).map_err(|e| CrawlError::GeneratorError(e))?;
        let input_file_base = input_file_base.unwrap();
        let mut new_file_name = String::from(input_file_base);
        new_file_name.push_str(".html");
        let output_path =

    ↪   output_directory.as_ref().join(&Path::new(&new_file_name));
        fs::write(&output_path, result)
            .map_err(|e|
    ↪   CrawlError::OutputFileError(format!("{:?}", e)))?;
        file_count += 1;
    } else {
        eprintln!("WARN: Encountered a non-template file (or non
    ↪   unicode path) during crawling the input directory {:?}",
    ↪   input_file);
    }
}
Ok(file_count)
}
```

Our crawling function, takes as input the input directory, the output directory, the template directory and the error strategy for the generator.

```
pub fn crawl_directories<P,Q,R>(
    output_directory: &P,
    input_directory: &Q,
    template_path: &R,
    err_strat: &GeneratorErrorStrategy
) -> Result<u32,CrawlError>
 where P : AsRef<Path>,
       Q : AsRef<Path>,
       R : AsRef<Path> {
let mut file_count = 0;
let input_files = input_directory.as_ref()
                  .read_dir()
                  .map_err(|_|
                        CrawlError::InputDirectoryError
                  )?;
for input_file in input_files {
   let input_file = input_file.map_err(|e|
↪  CrawlError::InputFileError(format!("{:?}", e)))?;
   let input_metadata = input_file.metadata().map_err(|e|
↪  CrawlError::InputFileError(format!("{:?}", e)))?;
   let input_file_name = input_file.file_name();
   let input_file_path = input_file.path();
   let input_file_extension =
↪  input_file_path.extension().and_then(|ext| ext.to_str());
   let input_file_base =
↪  input_file_path.file_stem().and_then(|stem| stem.to_str());
   if input_metadata.is_dir() {
       let dir_name = Path::new(&input_file_name);
       let new_output_dir = output_directory
                            .as_ref()
                            .join(&dir_name);
       let new_input_dir = input_directory
                           .as_ref()
                           .join(&dir_name);
       let n_count = crawl_directories(
           &new_output_dir,
           &new_input_dir,
           template_path,
           err_strat
       )?;
       file_count += n_count;
   } else if input_metadata.is_file() && (input_file_extension
↪  == Some("gop")) && (input_file_base.is_some()) {
       let input_text = {
          let mut temp = String::new();
          let mut file =
↪   File::open(input_file.path()).map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
          file.read_to_string(&mut temp).map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
          temp
       };
       let (template_name, mapping) =
↪   parse_source_string(&input_text).map_err(|e|
↪   CrawlError::ParseError(e))?;
       let template_path =
↪   template_path.as_ref().join(&Path::new(&template_name));
       let template_text = {
          let mut temp = String::new();
          let mut file = File::open(template_path).map_err(|e|
↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
          file.read_to_string(&mut temp).map_err(|e|
↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
          temp
       };
       let result = generate_output(
          template_text,
          mapping,
          err_strat
       ).map_err(|e| CrawlError::GeneratorError(e))?;
       let input_file_base = input_file_base.unwrap();
       let mut new_file_name = String::from(input_file_base);
       new_file_name.push_str(".html");
       let output_path =

↪   output_directory.as_ref().join(&Path::new(&new_file_name));
       fs::write(&output_path, result)
           .map_err(|e|
↪   CrawlError::OutputFileError(format!("{:?}", e)))?;
       file_count += 1;
   } else {
      eprintln!("WARN: Encountered a non-template file (or non
↪   unicode path) during crawling the input directory {:?}",
↪   input_file);
```

```
    }
}
Ok(file_count)
}
```

The errors produced by the crawler are as follows.

- **ParseError** - When a parser occurs
- **GeneratorError** - when a generator occurs
- **TemplateNotFound** - When a template is not found
- **InputDirectoryError** - When the input directory does not exist
- **OutputDirectoryError** - When the output directory does not exist

```
#[derive(Debug)]
pub enum CrawlError {
  ParseError(ParseError),
  GeneratorError(GeneratorError),
  TemplateNotFound(String),
  InputDirectoryError,
  OutputFileError(String),
  InputFileError(String),
}
```

Before we begin, let's set up a counter to enumerate the number of files converted.

```
let mut file_count = 0;
```

First, we'll extract all the files in the input directory.

```
let input_files = input_directory.as_ref()
                  .read_dir()
                  .map_err(|_|
                        CrawlError::InputDirectoryError
                  )?;
for input_file in input_files {
   let input_file = input_file.map_err(|e|
 ↪  CrawlError::InputFileError(format!("{:?}", e)))?;
   let input_metadata = input_file.metadata().map_err(|e|
 ↪  CrawlError::InputFileError(format!("{:?}", e)))?;
```

```
    let input_file_name = input_file.file_name();
    let input_file_path = input_file.path();
    let input_file_extension =
↪   input_file_path.extension().and_then(|ext| ext.to_str());
    let input_file_base =
↪   input_file_path.file_stem().and_then(|stem| stem.to_str());
    if input_metadata.is_dir() {
        let dir_name = Path::new(&input_file_name);
        let new_output_dir = output_directory
                             .as_ref()
                             .join(&dir_name);
        let new_input_dir = input_directory
                            .as_ref()
                            .join(&dir_name);
        let n_count = crawl_directories(
            &new_output_dir,
            &new_input_dir,
            template_path,
            err_strat
        )?;
        file_count += n_count;
    } else if input_metadata.is_file() && (input_file_extension
↪   == Some("gop")) && (input_file_base.is_some()) {
        let input_text = {
            let mut temp = String::new();
            let mut file =
↪   File::open(input_file.path()).map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
            file.read_to_string(&mut temp).map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
            temp
        };
        let (template_name, mapping) =
↪   parse_source_string(&input_text).map_err(|e|
↪   CrawlError::ParseError(e))?;
        let template_path =
↪   template_path.as_ref().join(&Path::new(&template_name));
        let template_text = {
            let mut temp = String::new();
            let mut file = File::open(template_path).map_err(|e|
↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
            file.read_to_string(&mut temp).map_err(|e|
↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
            temp
        };
        let result = generate_output(
            template_text,
            mapping,
            err_strat
        ).map_err(|e| CrawlError::GeneratorError(e))?;
        let input_file_base = input_file_base.unwrap();
        let mut new_file_name = String::from(input_file_base);
        new_file_name.push_str(".html");
        let output_path =

↪   output_directory.as_ref().join(&Path::new(&new_file_name));
        fs::write(&output_path, result)
            .map_err(|e|
↪   CrawlError::OutputFileError(format!("{:?}", e)))?;
        file_count += 1;
    } else {
        eprintln!("WARN: Encountered a non-template file (or non
↪   unicode path) during crawling the input directory {:?}",
↪   input_file);
    }
}
```

For each file, we need to check its metadata.

```
let input_file = input_file.map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
let input_metadata = input_file.metadata().map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
let input_file_name = input_file.file_name();
let input_file_path = input_file.path();
let input_file_extension =
↪   input_file_path.extension().and_then(|ext| ext.to_str());
let input_file_base =
↪   input_file_path.file_stem().and_then(|stem| stem.to_str());
```

Now our next action is dependent on the type of entry - we'll need to do different things based on whether we find a file or a directory.

```
if input_metadata.is_dir() {
    let dir_name = Path::new(&input_file_name);
    let new_output_dir = output_directory
                         .as_ref()
                         .join(&dir_name);
    let new_input_dir = input_directory
                        .as_ref()
                        .join(&dir_name);
    let n_count = crawl_directories(
        &new_output_dir,
        &new_input_dir,
        template_path,
        err_strat
    )?;
    file_count += n_count;
} else if input_metadata.is_file() && (input_file_extension ==
↪   Some("gop")) && (input_file_base.is_some()) {
    let input_text = {
       let mut temp = String::new();
       let mut file = File::open(input_file.path()).map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
       file.read_to_string(&mut temp).map_err(|e|
↪   CrawlError::InputFileError(format!("{:?}", e)))?;
       temp
    };
    let (template_name, mapping) =
↪   parse_source_string(&input_text).map_err(|e|
↪   CrawlError::ParseError(e))?;
    let template_path =
↪   template_path.as_ref().join(&Path::new(&template_name));
    let template_text = {
       let mut temp = String::new();
       let mut file = File::open(template_path).map_err(|e|
↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
       file.read_to_string(&mut temp).map_err(|e|
↪   CrawlError::TemplateNotFound(format!("{:?}", e)))?;
       temp
    };
    let result = generate_output(
       template_text,
       mapping,
       err_strat
    ).map_err(|e| CrawlError::GeneratorError(e))?;
    let input_file_base = input_file_base.unwrap();
    let mut new_file_name = String::from(input_file_base);
    new_file_name.push_str(".html");
    let output_path =

↪   output_directory.as_ref().join(&Path::new(&new_file_name));
    fs::write(&output_path, result)
        .map_err(|e|
↪   CrawlError::OutputFileError(format!("{:?}", e)))?;
    file_count += 1;
} else {
   eprintln!("WARN: Encountered a non-template file (or non
↪   unicode path) during crawling the input directory {:?}",
↪   input_file);
}
```

Now, if the file is a directory, we do a recursive call, appending the directory name to the input path and output path

```
let dir_name = Path::new(&input_file_name);
let new_output_dir = output_directory
                     .as_ref()
                     .join(&dir_name);
let new_input_dir = input_directory
                    .as_ref()
                    .join(&dir_name);
let n_count = crawl_directories(
    &new_output_dir,
    &new_input_dir,
    template_path,
    err_strat
)?;
file_count += n_count;
```

On the other hand, if the file is just a file, we first need to read the file.

```
let input_text = {
   let mut temp = String::new();
   let mut file = File::open(input_file.path()).map_err(|e|
 ↪  CrawlError::InputFileError(format!("{:?}", e)))?;
   file.read_to_string(&mut temp).map_err(|e|
 ↪  CrawlError::InputFileError(format!("{:?}", e)))?;
   temp
};
```

Now we'll run the parser on this text.

```
let (template_name, mapping) =
 ↪  parse_source_string(&input_text).map_err(|e|
 ↪  CrawlError::ParseError(e))?;
```

Now we need to read the template to a string.

```
let template_path =
 ↪  template_path.as_ref().join(&Path::new(&template_name));
let template_text = {
   let mut temp = String::new();
   let mut file = File::open(template_path).map_err(|e|
 ↪  CrawlError::TemplateNotFound(format!("{:?}", e)))?;
   file.read_to_string(&mut temp).map_err(|e|
 ↪  CrawlError::TemplateNotFound(format!("{:?}", e)))?;
   temp
};
```

With the template and the mapping, we can run the generator.

```
let result = generate_output(
   template_text,
   mapping,
   err_strat
).map_err(|e| CrawlError::GeneratorError(e))?;
```

Before we write this to the output directory, we need to construct a new name for the file.

```
let input_file_base = input_file_base.unwrap();
let mut new_file_name = String::from(input_file_base);
new_file_name.push_str(".html");
```

Finally, we can write this to the output directory.

```
let output_path =
    output_directory.as_ref().join(&Path::new(&new_file_name));
fs::write(&output_path, result)
    .map_err(|e| CrawlError::OutputFileError(format!("{:?}",
↪   e)))?;
file_count += 1;

Ok(file_count)
```

6 Error Strategy

Now for the final part of the application - implementing the error strategy from before.

Before we do anything, we'll need to extend the capabilities of a prior structure - specifically the GeneratorErrorCoreStrategy, and the capability to parse the element from a string.

```
impl FromStr for GeneratorErrorCoreStrategy {
    type Err = ();
    fn from_str(src: &str) ->
↪   Result<GeneratorErrorCoreStrategy, ()> {
        return match src {
            "fail" => Ok(GeneratorErrorCoreStrategy::Fail),
            "ignore" => Ok(GeneratorErrorCoreStrategy::Ignore),
            x => {
                if let Some(ind) = src.find("=") {
                   if ind + 1 < src.len() {
                       let (txt, rem) = src.split_at(ind+1);
                       if txt == "fixed=" {

↪   Ok(GeneratorErrorCoreStrategy::Fixed(rem.to_string()))
                       } else {
                           Err(())
                       }
                   } else {
                       Err(())
                   }
                } else {
                  Err(())
                }
            },
        };
    }
}
```

As you can see, we're referencing the `FromStr` trait which we'll need to import.

```
use std::str::FromStr;
```

Now let's just quickly add some tests to verify this actually works.

```
#[test]
fn from_st_works() {
  assert_eq!(GeneratorErrorCoreStrategy::from_str("ignore"),
↪  Ok(GeneratorErrorCoreStrategy::Ignore));
  assert_eq!(GeneratorErrorCoreStrategy::from_str("fail"),
↪  Ok(GeneratorErrorCoreStrategy::Fail));

↪  assert_eq!(GeneratorErrorCoreStrategy::from_str("fixed=missing"),
↪  Ok(GeneratorErrorCoreStrategy::Fixed("missing".to_string())));
}
```

Okay, now onto the topic of determining an error response strategy.

We'll be doing this by splitting the concerns into two separate components - first identifying the core strategy and then identifying the use of a default strategy or not.

First for the core strategy, we'll set a default and then populate it.

```
let mut opt_strat =
↪  generator::GeneratorErrorCoreStrategy::Fail;
```

Using the from string implementation we described earlier, we can parse this as follows.

```
ap.refer(&mut opt_strat)
  .add_option(&["-e", "--error"],
              argparse::Store,
              "Fail on the first undefined parameter");
```

For the default strategy we'll be using an optional value which we'll try and populate. If it isn't populated then we'll know that there is no default strategy.

```
let mut def_strat = None;
```

Once again, for the default we'll just try and populate the string.

```
ap.refer(&mut def_strat)
  .add_option(&["-d", "--default"],
              argparse::StoreOption,
              "Additional mapping for storing default values.
↪  If not specified, the environment variable
↪  GOP_HTML_DEFAULTS if defined, is used as a default.");
```

If none was provided we'll try and retrieve it from the environment under the key GOP_HTML_DEFAULTS.

```
let def_strat = def_strat.or_else(||
↪  env::var("GOP_HTML_DEFAULTS").ok());
```

Finally, we can construct the error strategy based on whether a default is provided.

```
let err_strat = match def_strat {
   None =>
      generator::GeneratorErrorStrategy::Base(opt_strat),
   Some(path) => {
      let mapping = {
          let def_path = Path::new(&path);
          if let Ok(mut file) = File::open(&def_path) {
             let mut def_source = String::new();
             if let Ok(_count) = file.read_to_string(&mut
↪  def_source) {
                 parser::parse_source_string(&def_source).ok()
             } else {
                 None
             }
          } else {
              None
          }
      };
      match mapping {
        Some((name, map)) =>
            generator::GeneratorErrorStrategy::Default(map,
↪  opt_strat),
        None => {
            eprintln!("Encountered error while reading default
 ↪  mapping at {:?}.", path);
            generator::GeneratorErrorStrategy::Base(opt_strat)
        }
      }
   }
};
```

Now all we've got to do is retrieve the mapping.

```
let def_path = Path::new(&path);
if let Ok(mut file) = File::open(&def_path) {
    let mut def_source = String::new();
    if let Ok(_count) = file.read_to_string(&mut def_source) {
        parser::parse_source_string(&def_source).ok()
    } else {
        None
    }
} else {
    None
}
```