Tar Heel Extended Regular Expressions - `thegrep`

PS05 - COMP590 - Spring 2019

Overview

The classic `grep` text pattern-matching search tool was originally authored in 1974 by Ken Thompson. It is widely used today. This command-line utility marries the theory of regular languages and automata with the practical application of efficiently searching bodies of text.

Regular expressions can readily be represented as simple state machines often illustrated as transition diagrams or annotated digraphs. These concepts will be introduced in lecture and are covered in full depth in COMP455. The general idea, though, is that a target regular expression pattern can be transformed into a "machine" whose input is textual characters and whose purpose is to determine whether the target pattern exists in the text.

Regular expressions are a wonderful little language and the first steps to begin implementing a regular expression engine are similar to those taken for `thdc` and `thdc`. You will need to tokenize and parse a regular expression pattern into an *abstract syntax tree or AST* (like the `Expr` tree in `thbc`). In the *next* parts of this problem set, you will convert the parsed AST into a non-deterministic finite automaton (NFA). That's just the fancy term for the "machine" described in the previous paragraph. Finally, you will implement the algorithm to feed lines of files into your machine and determine if any match by *simulating* the NFA. For now, your concern is tokenization and parsing of the regular expression pattern.

In this problem set you will begin nearly from scratch, establish command-line options and input parsing, and then implement a tokenizer and parser. The lexeme definitions, `Token` enum definition, grammar, and `AST` enum are provided on the following page. Required command-line options and expected outputs are specified thereafter.

Lexemes of `thegrep`

⟨*LParen*⟩ ::= '`(`'

⟨*RParen*⟩ ::= '`)`'

⟨*UnionBar*⟩ ::= '`|`'

⟨*KleeneStar*⟩ ::= '`*`'

⟨*AnyChar*⟩ ::= '`.`'

⟨*Char*⟩ ::= any character except '`(`', '`)`', '`|`', '`.`', '`*`'

Grammar of `thegrep`

The terminals are specified in the definitions above. Note the challenge of parsing `Catenation` is in knowing whether the next character is actually the start of an `Atom` or not. You should think about and understand *why* that is.

⟨*RegExpr*⟩ ::= ⟨*Catenation*⟩ (`UnionBar` ⟨*RegExpr*⟩)?

⟨*Catenation*⟩ ::= ⟨*Closure*⟩ ⟨*Catenation*⟩?

⟨*Closure*⟩ ::= ⟨*Atom*⟩ `KleeneStar`?

⟨*Atom*⟩ ::= `LParen` ⟨*RegExpr*⟩ `RParen` | `AnyChar` | `Char`

`Token` and `AST` Enumeration Types

You should use the following definition with exactly matching names of tokens and abstract syntax tree nodes. You will ultimately need to print these out using debug formatting for autograding. You'll also need to compare them with expected values in unit tests. As such, it will also need to derive `Debug` and `PartialEq`.

```
pub enum Token {
    LParen,
    RParen,
    UnionBar,
    KleeneStar,
    AnyChar,
    Char(char),
}

pub enum AST {
    Alternation(Box<AST>, Box<AST>),
    Catenation(Box<AST>, Box<AST>),
    Closure(Box<AST>),
    Char(char),
    AnyChar
}
```

Command Line Options

Running `cargo run -- --help` should display a help message similar to the following output. You will need to implement the flags `t` and `p` with their respective `tokens` and `parse` long forms. You are encouraged to rely upon the `structopt` crate as shown in lecture.

The input regular expression pattern you are tokenizing and parsing will be the *positional argument* following any flags. Please refer to the `structopt` crate's documentation for examples of how to do achieve this: https://docs.rs/structopt/

```
thegrep 1.0.0
Tar Heel egrep

USAGE:
    thegrep [FLAGS] <pattern>

FLAGS:
    -h, --help       Prints help information
    -p, --parse      Show Parsed AST
    -t, --tokens     Show Tokens
    -V, --version    Prints version information

ARGS:
    <pattern>    Regular Expression Pattern
```

The `--tokens` Flag

Running your program with `--tokens` should result in each of the `Token` values' `Debug` representation being printed individually on each line.

Example usage:

```
$ cargo run -- --tokens 'ab|().*'
Char('a')
Char('b')
UnionBar
LParen
RParen
AnyChar
KleeneStar
```

The `--parse` Flag

Running your program with `--parse` should result in the parsed `AST` value's `Debug` representation being printed.

If an error is encountered during parsing, please print any error message to `stderr` using the `eprintln!` macro with the prefix `thegrep:` before the actual error message.

Example usage (*Note: new lines and indentation are NOT expected and were added manually to this document to help improve legibility. You should just use the default* `Debug` *formatting.*):

```
$ cargo run -- --parse 'a.*'
Catenation(
    Char('a'), Closure(AnyChar))

$ cargo run -- --parse 'abc'
Catenation(
    Char('a'),
    Catenation(
        Char('b'),
        Char('c')))

$ cargo run -- --parse 'a|b|c'
Alternation(
    Char('a'),
    Alternation(
        Char('b'),
        Char('c'))

$ cargo run -- --parse '(ab)*'
Closure(
    Catenation(
        Char('a'),
        Char('b')))

$ cargo run -- --parse 'b(oo*|a)m'
Catenation(
    Char('b'),
    Catenation(
        Alternation(
            Catenation(
                Char('o'),
                Closure(
                    Char('o'))
            ),
            Char('a')
        ),
        Char('m')))
```

Design Documentation

You will need to add a `README.md` file to the root folder that describes your overall design. This document is in markdown format (like the GRQs) and will show up automatically on your GitHub repository. You should include any design decisions you're particularly proud of as well as any notes you believe would benefit the graders to be aware of. For *pair* submissions, please add a section to this document describing what you contributed and how you collaborated.

Getting Started

GitHub classroom starter URL: https://classroom.github.com/g/9iLvreB7

Please follow the link above to setup your repository for this problem set. If you are working as a team, agree upon who will establish the team and a team name on GitHub classroom *first* so that the other person can join the team *second*. **Do not join anyone's team unless you have communicated about doing so ahead of time.** If you are working alone, it appears you'll still need to come up with a team name. Once your repository is ready, you'll want to find its Clone link and be sure you choose the SSH option that begins with `git@github.com:comp590-...`. Copy that link to your clipboard.

While logged in to your VM in a terminal, issue the following commands and replace the words in <tags> with the values specific to you:

```
$ cd $HOME
$ git clone <paste>
$ cd <repo>
```

You should go ahead and edit `Cargo.toml` to have your name(s) in it and fill in the honor pledge in `src/main.rs`.

Grading Rubric Breakdown

Autograding Levels

Basic tests for all levels will be released as soon as they're ready. On the deadline, as per the `thdc` problem set, a complete suite of tests will be added.

1. 10 points - Useful Command-line Flag Support for `--help`
2. 30 points - Tokenization Debug Output via `-t` or `--tokens`
3. 50 points - Parsing Debug Output via `-p` or `--parse`

Hand-graded Points

1. 10 points - README Design Documentation, Code Style & Documentation, Appropriate Use of Multiple Files to Organize Project
2. 10 points - Unit Tests