CSC 230 Project 6

Regular Expression Matcher

For this project, you're going to be writing a program called `regular`. You will be able to run the program, giving it a regular expression on the command line. It will then read lines of text from an input file or from standard input, printing out lines that match the pattern and highlighting the parts of each line that match the pattern.

For example, test input 14 uses the pattern `a(bc)*d`. This says to match any text containing an `a`, followed by any number of occurrences of `bc` (including zero occurrences), followed by `d`. The input file, `input-14.txt` contains the following lines. The first four of these match the pattern, since they contain a string that stats with `a`, followed by zero or more occurrences of `bc`, followed by `d`. The rest of the lines don't match the pattern.

```
abcd
ad abcdefghijk bcdefg
abcbcbcbcd
xyz abcd 123
abbbbd
accccd
acbcbd
xbcbcx
a bc d
```

You can run the program as follows, giving it this pattern on the command line and reading input from standard input (redirected from the input-14.txt file). As output, it prints each input line that contains a match, with the matches highlighted in red.

```
$ ./regular 'a(bc)*d' < input-14.txt
abcd
ad abcdefghijk bcdefg
abcbcbcbcd
xyz abcd 123
```

Sample execution with matches highlighted

Notice that we're putting single quotes around the pattern. We're going to need to do this for most patterns, since some of the special characters used in regular expressions are also special characters for the shell. Putting them in single quotes protects them from special interpretation by the shell.

Our program will use a little inheritance hierarchy to implement a significant part of the regular expression syntax. This is a common way of describing and matching text patterns, available in lots of different shell commands and programming languages.

You will be developing this project in the p6 directory of your csc230 git repo, and, as usual,

you'll submit by pushing your changes up to the NCSU github repo before the due date.

We're providing you with a starter that includes lots of files to help you organize and test your program. See the Getting Started section for instructions on how to get the starter and unpack it into your repo.

This project supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 6

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you to follow these guidelines as you're planning and implementing your solution.

Requirements

Program Execution

The `regular` program can be run with either one command-line argument or with two. Its first command-line argument is always the pattern it's supposed to search for. An optional, second command-line argument gives the name of the file from which it's supposed to read and match lines. If only one command-line argument is given, it will read and match lines from standard input. For example, if run as follows, the program will read lines from the file `test-11.txt` and print out those that match the pattern, `ab*c`:

```
$ ./regular 'ab*c' test-11.txt
```

Run as follows, the program will expect input lines entered from the terminal:

```
$ ./regular 'ab*c'
```

If the user attempts to run the program with invalid arguments (e.g., too many or too few), it should print the following usage message to standard error and then exit with an exit status of `EXIT_FAILURE`.

```
usage: regular <pattern> [input-file.txt]
```

If the program is given an input file that it can't open, it will print the following message to standard error (where `filename` is the name of the file it wasn't able to open) and terminate with exit status, `EXIT_FAILURE`.

```
 Can't open input file: filename
```

If the given pattern isn't a valid regular expression, it will print the following message to standard error and exit with a status of `EXIT_FAILURE`. The program should try to open the input file before trying to parse the pattern, so if they're both bad, it will just report the `Can't open input file` message.

```
Invalid pattern
```

Program Input

After it's started, `regular` just needs to read lines from its input until it reaches the end-of-file. It needs to be able to handle input lines of up to 100 characters in length (not counting the newline character at the end of each input line). If the program encounters a line that's too long, it should print the following message to standard error and terminate with an exit status of `EXIT_FAILURE`. If this happens, the program will have already printed output for all previous input lines.

```
Input line too long
```

Program Output

The program should print to standard output any line that contains a match for the pattern, and omit lines that don't contain a match.

For each line containing a match of the pattern, the program should highlight the matching portions by printing all characters that are part of a match in red. There may be multiple matches for each input line, and there may even be some individual characters that are part of multiple matches. Any character that's part of any match will need to be printed in red.

There are ANSI escape codes that can be inserted to change the text color as necessary. Printing the sequence of characters, ESC (ASCII code 27 in decimal), '[', '3', '1' then 'm' switches the text color to red. When you no longer need to print in red, you can print the character sequence, ESC, '[', '0', 'm', to return the print color to the default.

If multiple consecutive characters need to be highlighted, just change the color to red before the first character and back to the default after the last one (as opposed to changing to red and back to the default before and after each highlighted character). This won't make a difference in how your output looks, but it will make sure your output matches the expected output.

Regular Expressions Patterns

In the regular expression syntax, a pattern consists of ordinary characters (like 'a' and '5') that just match occurrences of themselves. A pattern can also contain metacharacters, that match things other than themselves, that help to control how the regular expression is parsed, or that determine how parts of it behave. A pattern can be:

- An ordinary character, any printable character other than newline and characters in the

set `.^$*?+|()[{`.
An ordinary character matches any occurrence of itself, anywhere in the string. For example, the pattern `a` will match the 'a' in string "abc", "cba" or three places in "xxxaaayyy", but nothing in "xyz".

- The `.` symbol.
  This metacharacter matches a single occurrence of any character. So, the pattern `.` will match every character in the string "abc", "xyz" or even ".", but it won't match the empty string.

- The start anchor, `^`.
  This metacharacter matches the start of the line, a location right before the first character on the line. This can be used to describe patterns that will only match things at the beginning of the line.

- The end anchor, `$`.
  This metacharacter matches the end of the line, a location right after the last character on the line. This can be used to make patterns that only match things at the end of the line, or used along with `^`, to make patterns that have to match everything on the line.

- A character class, a sequence of characters inside square brackets (not including the character `]` or newline).
  This matches any one character given in the sequence. For example, the pattern `[abc]` will match any one occurrence of the letter `a` or the letter `b` or the letter `c`. This gives us a way of matching characters from just about any set we need, like the set of decimal digits, `[0123456789]`, and, since just about any character can appear inside square square brackets, it gives us a ... umm ... clever way to literally match symbols that would otherwise be interpreted as metacharacters, like `[.]` or `[$]`. It's OK for for the same character to be given more than once when defining a character class (e.g., `[aabc]`); there's no good reason to do this, but it doesn't make the pattern invalid.

- Any pattern inside parentheses.
  As with mathematical expressions, we can use parentheses to control how a regular expression is parsed. Although it uses building blocks from later in this list, a simple example would be `ab*` which would match an occurrence of `a` followed by any number of repeated occurrences of `b`, while `(ab)*` would match any number of repeated occurrences of 'ab'.

- A pattern, *p*, followed by `*`.
  This will match zero or more consecutive occurrences of anything *p* matches. For example, `b*` would match the middle of the strings "abc", "abbbc" or even "ac" (zero occurrences of b, so, by itself, `b*` isn't particularly useful).

- A pattern, *p*, followed by `+`.
  This will match one or more consecutive occurrences of anything *p* matches.

- A pattern, *p*, followed by `?`.

This will match zero or one occurrences of anything *p* matches, so a pattern followed by a question mark is like an optional match in a pattern.

- Any two consecutive patterns, *p1* and *p2*.
  This represents concatenation, and matches anything that can be matched by *p1* followed immediately by anything that matches *p2*. For example, `[ab][cd]` matches "ac", "ad", "bc" or "bd".

- Any two patterns, *p1* and *p2* separated by `|`.
  This is called alternation; it matches anything matched by either *p1* or *p2*. So, "cat|dog" will match "cat" or "dog".

The description above is inductive; it describes how to make the smallest regular expressions (the base cases), then how these can be combined into more complex regular expressions. This is how regular expressions (or programming languages) are normally described, starting with small bits of syntax and then showing how it can be composed. Anything that's inconsistent with the description above (e.g., `*` or `[abc`) would be considered an invalid pattern.

The description above is also ordered by precedence. Everything from ordinary symbols up to parentheses is at the highest level of precedence. The repetition operators, `*`, `+` and `?` are at the next highest level. Concatenation is at the next highest level and alternation is at the lowest precedence. Later, when we talk about building the parser for regular expressions, we'll need to keep our parsing code organized so it respects the precedence described here.

If you're unfamiliar with regular expressions, you'll find a lot of useful resources online to help you understand the syntax and what it means. The Wikipedia page for regular expressions is fairly extensive. If you use outside resources like this to supplement your understanding of regular expressions, keep in mind that we're implementing a subset of what most tools support, and there are a few thing that we're doing slightly differently. Try to use the description above as a guide for what your program is actually expected to do.

Regular Expression Examples

This section describes a few examples of regular expressions, some straightforward and some that might be considered tricky or special cases. As we run into situations that seem non-obvious or that require additional explanation, we can extend this list of examples.

- `[ABCDEFGHIJKLMNOPQRSTUVWXYZ][abcdefghijklmnopqrstuvwxyz]+`
  This would match any capitalized word, like "David" or "Knoxville", but it wouldn't match single-character words, like "I" or words in all caps like "NASA".

- `^[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ]+$`
  This would match non-empty lines consisting of only letters and spaces.

- `[0123456789]+`
  This would match any decimal integer.

- `[0123456789]+[.][0123456789]+`
  This would match any real number, given with a whole number and a fractional part.

- `[0123456789]+?[.][0123456789]+`
  This would match any real number, with an optional whole number part, so it could match 3.14 and .14. We could have done this with `*` instead of `+?`, but I wanted to show that you can stack up uses of the repetition metacharacters, since they work for any pattern, even patterns that already use repetition.

- `^a`
  This pattern would match the start of the line, followed by the character 'a', so it would match any line starting with `a`, like "a" or "abc".

- `^$`
  This would only match empty lines.

- `^..........$`
  This would only match lines containing exactly 10 characters.

- `^^a`
  This pattern would also match any line starting with `a`. The first `^` matches the start of the line, but it doesn't match any actual characters, so the second `^` can still match the start of the line. After matching the first `^`, you're still at the start of the line, so the second '^' can still match. Compare this to the next example.

- `.^`
  In our program, this would never match any lines. The start-of-line anchor couldn't match anything that came after a symbol. It's still a valid pattern, since we can parse it, but it couldn't match anything.

- `^*a`
  This is a stupid pattern. Putting the * after the `^` will match any number of occurrences of the start-of-line, including zero occurrences. So, this will match any line that contains an `a` anywhere, not necessarily at the start.

- `$^$^$^`
  It might seem like this would never match anything, but there's one case where it will match. Think about it for a moment (if you want to). It's an empty line. For an empty line, the start and end location are the same, so any sequence of `^` and `$` will match it.

Although some of these examples might seem like special cases, I don't think I had to do anything special in my program to handle any of them. The technique I used for matching patterns (described in the design section) took care of all these situations automatically.

Extra Credit, Extended Regular Expression Syntax

For this assignment, we've simplified some parts of the regular expression syntax, to make

things a little easier. If you'd like some extra credit, you can support a more flexible syntax for character classes and the repetition operation.

Often, character classes support a more flexible syntax. For 8 points of **extra credit**, have your character classes support the following:

- Putting a `^` at the start of a non-empty character class inverts it, causing it to match any character other than those specified in the character class. For example, `[^abc]` will match any character other than `a`, `b` or `c`. To get the inverse behavior, something has to follow the `^`. The pattern `[^]` should just match the symbol `^` literally (it looks like this is a little different from what some sandard regular expression libraries do).
- Putting a `-` between two characters, like c_1-c_2 in the character class is a short-hand notation for all character codes for characters from c_1 up to and including c_2. If the character code for c_1 is greater than the code for c_2, the pattern is invalid, (so, you should handle `a-z` but not `z-a`). If the - occurs at the start or end of the character class (or after the initial `^`), it should be interpreted literally. So, `[^a-z]` will match anything that's not a lower-case letter, but the pattern `[a-]` will just match `a` and `-`, the pattern `[-z]` will just match a `-` and a `z`, and the pattern `[^-z]` will match anything other than a `-` and a `z`. It's OK to have multiple ranges in a character class, so `[0-9a-zA-Z]` should match any letter or decimal digit.

Regular expressions often support a more flexibility repetition syntax in addition to the one required for this assignment. For 8 points of **extra credit**, support patterns of the form:

- A pattern *p* followed by `{n,m}`, where `n` an `m` are decimal integers.
  This matches between `n` and `m` consecutive occurrences of patterns matched by *p* (inclusive). So, `[ab]{5,10}` will match any sequence of `a` and `b` characters between 5 and 10 characters long. If `n` is larger than `m`, the pattern should be considered invalid.
- A pattern *p* followed by `{,m}`.
  This matches between zero and `m` occurrences of anything matched by *p*.
- A pattern *p* followed by `{n,}`.
  This matches `n` or more occurrences of anything matched by *p*.

In this repetition syntax, a pattern would be considered invalid if the value of `n` was greater than the value of `m` (e.g., `{5,3}`), or if the strings given for `n` and `m` were non-empty but not legal integer values (e.g., `{abc,xyz}`).

Design

There are a few different ways we could write code for regular expressions. For this project, we've tried to choose a technique that's not too difficult to implement and that can be easily extended with an inheritance hierarchy.

Starter and Source Code Organization

This starter for this project is organized into three components:

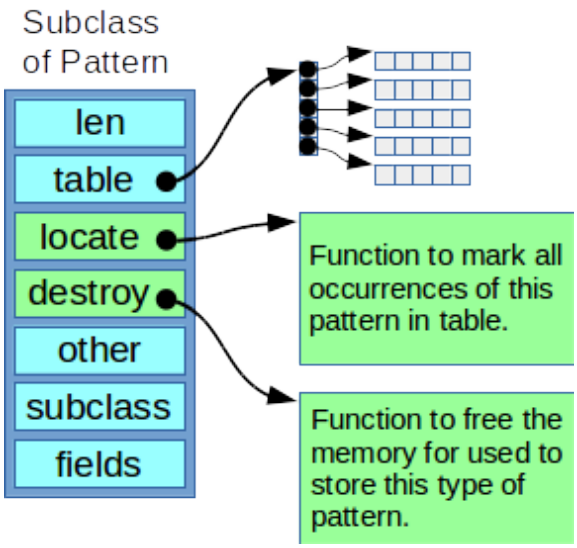
- `pattern.c` / `pattern.h`
  The pattern component implements the inheritance hierarchy used for regular expressions. It defines the abstract superclass for a Pattern as well as different concrete subclasses for represent various parts of regular expression syntax.
  The starter includes a partial implementaton for the pattern component. It defines the pattern superclass, along with a few functions that serve as methods for the class. It also includes concrete implementations of pattern for matching individual symbols (SymbolPattern) and for matching concatenated patterns (BinaryPattern). As you add support for different regular expression syntax, you'll create new struct and function definitions in the pattern implementation file, exposing prototypes (and comments) for their constructors in the header.

- `parse.c` / `parse.h`
  The parse component parses the text of a regular expression and turns it into a collection of Pattern objects that represent it.
  The starter includes a partial implementation of this component. It has all the parsing functions you'll need, but you'll have to add code inside these functions to be able to parse all the building blocks of pattern syntax.

- `regular.c`
  This component contains the main() function. It's responsible for handling command-line arguments, using the parser to build a representation for the regular expression, reading input lines, using the Pattern representation to find places where the pattern matchs the an input line, and printing out the lines with the matches highlighted in red.
  In the starter, the main function contains some sample code to show how to parse and use regular expressions. This sample code is written with some hard-coded regular expressions and strings. As you complete your implementation, you'll probably delete most of the code and replace it with your own code.

Pattern Inheritance Hierarchy

We're going to implement an inheritance hierarchy for patterns, with different subclasses for different types of patterns. The superclass, Pattern has already been implemented for you. It contains an arbitrary-sized table called the *match table*. This table is represented as an array of pointers to arrays of boolens. The match table is used to record the places where the pattern matches a substring of the input string. Each time we apply the pattern to a different string, we allocate and initialize a new match table, with a size that's determined by the current input string.



Organization of a Pattern subclass

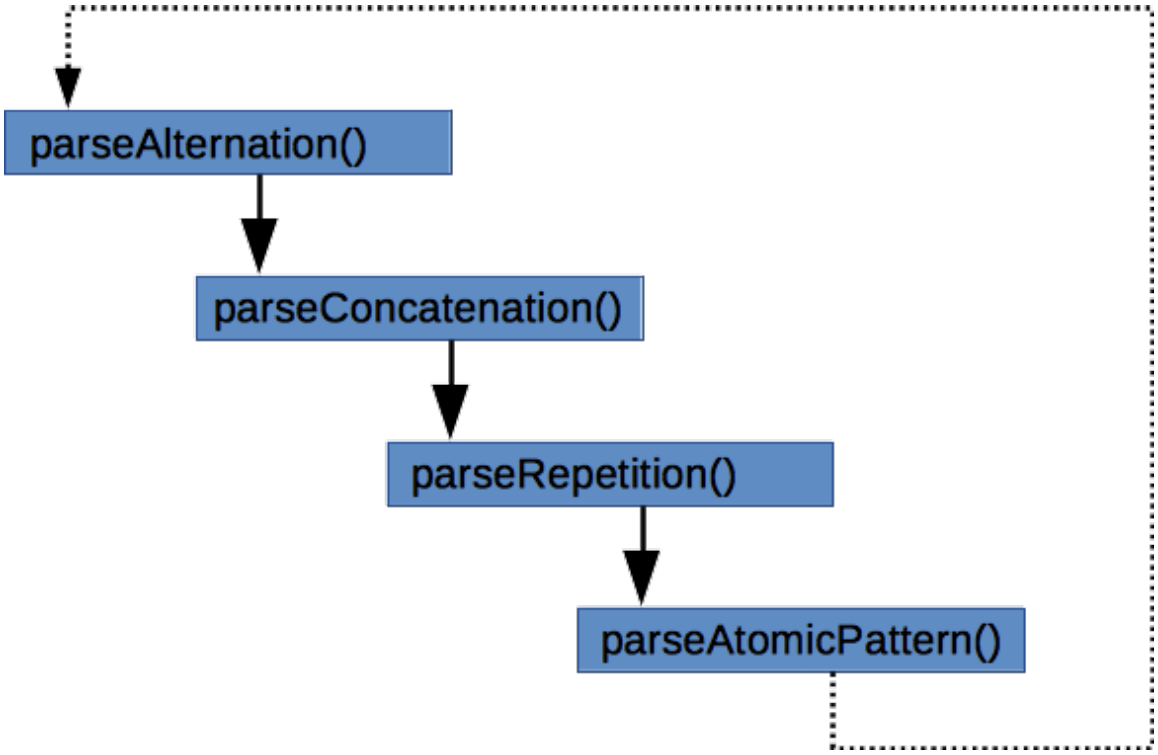
The Pattern superclass also contains two function pointers. We'll use these like override-able methods. The locate field points to a functions that finds all substrings in the current string that match the pattern (see below). Every pattern also has a function pointer pointing to its destroy function. Calling this function will free all the memory used by a pattern, including calling destroy to free any memory use by its sub-patterns.

Like we did in class, we'll make subclasses of Pattern by making structs that start with the same layout of fields in memory, including any function pointers we need to do the job of an overridable method. After these common fields, the subclass can add any additional fields it needs. Code that's made to work with the superclass, Pattern, will be able to use the common fields without worrying about what comes afterward in memory. Code that's specific to a subclass can cast a Pattern pointer to a more specific type and then access the fields specific to the particular subclass.

You'll need to implement some additional Pattern subclasses to finish the application. You can decide for yourself what they should contain and what you want to name them. In your completed implementation, you need to have at least **two additional struct types** used for implementing subclasses of Pattern. You can probably re-use some types to match multiple kinds regular expressions; some of them could be easily adapted to do multiple jobs by just changing the function pointed to by the locate field.

Parsing

The starter includes some of the code you'll need for parsing the regular expression given on the command line. It's a typical recursive descent parser. It starts with a function to parse the lowest-precedence parts of a regular expression, multiple smaller regular expressions connected with alternation (vertical bar). This calls a function to take care of the next-lowest precedence, concatenation. At the bottom, parseAtomicExpression() takes care of the highest-precedence syntax, ordinary symbols, anchors, character classes and regular expressions inside parentheses.

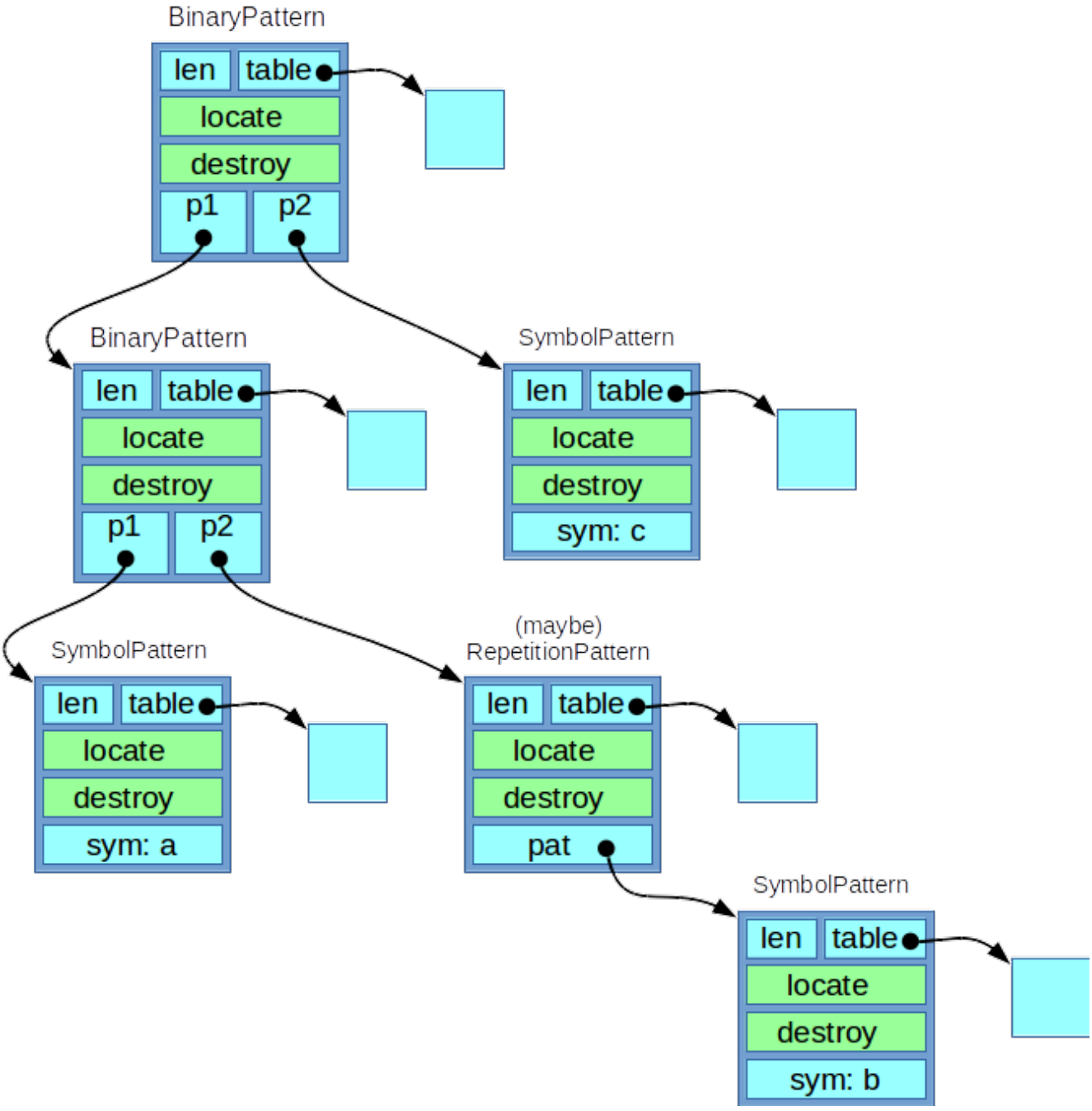


Functions for parsing regular expressions

The dashed arrow in the figure above shows how you'll handle parentheses. After the start of a parenthetical expression, the parseAtomicExpression() function will just start over with parseAlternation() to parse the expression inside parentheses.

The parser in the starter is incomplete. It has the four parsing functions shown above, and it knows how to parse ordinary symbols and concatenation. You'll need to add code to most of the parsing functions to get it to parse all the types of regular expressions we want to support.

The job of the parser is to build a tree of Pattern objects representing the regular expression you've parsed. Your `pattern` component will actually implement these objects, exposing just a constructor for each type of object. You get to decide what object types you're going to use, but the following figure shows how you might parse an expression like "ab+c". At the leaves, we have instances of SimplePattern for matching occurrences of `a`, `b` and `c`. As the parent of the the middle SimplePattern, I've drawn an object, RepetitionPattern, responsible handling the `+` operator by matching one or more occurrences of its sub-pattern `b`. I put the word "maybe" above this object because that particular object is your job to design. It makes sense to me to have a type of pattern for handling repetition (that's what I did), but maybe you'll think of a different way.



Representation for the pattern, "ab+c"

Higher in the tree, we have instances of BinaryPattern, used to handle concatenation of subpatterns. This type of object and the code to parse it is already provided for you in the starter.

Matching Patterns

A pattern's locate function is used to find all occurrences of the pattern in some given string. It records all such occurrences in its match table. Consider a substring that goes from the character at index *begin* up to but not including the index at *end*. If the pattern can match

this substring, then the pattern's locate function should set `table[ begin ][ end ]` to true. For example, if you had a pattern like `a+` and the current string was "baabab", then the pattern should set `table[ 1 ][ 2 ]` to true, along with `table[ 2 ][ 3 ]`, `table[ 1 ][ 3 ]` and `table[ 4 ][ 5 ]`. These elements of the match table correspond to the substrings of "baabab" consisting of one or more 'a' characters. For example, `table[ 1 ][ 3 ]` corresponds to the "aa" substring near the start, it starts at character 1 and ends right before character 3.

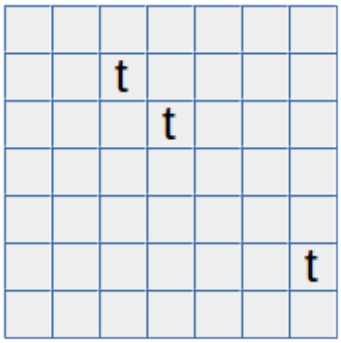
After all the matches for a pattern have been found, they can be quickly looked up by using the table. There's a function that does this, matches(), defined in the pattern component. It's like an instance method defined for pattern, you give it a pointer to a pattern and a *begin* and *end* location, and it returns an element of that pattern's match table.

Each pattern has its own table, to record places where that pattern matches the current input string. The starter gives you a few functions to help manage the table for each pattern. The freeTable() function discards the memory for a previously allocated table, useful when you're destroying a pattern or preparing to match the pattern against a different string. There's also an initTable() function, to make a new table for matching the given pattern against a new string.

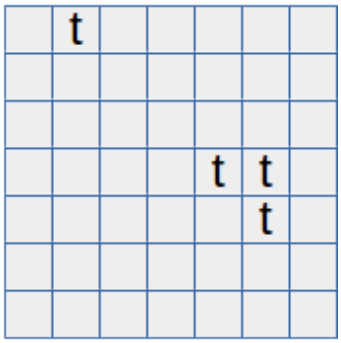
The match table always needs to be one column wider than the length of the current string, and one row taller. This lets it record matches for any substring of the current input string. On an n-character string, for example, `table[ 0 ][ n ]` represents the entire string. The table can also record matches for zero-length strings. This is necessary, since some patterns, like `^` or `$` or `b*` that can match a zero-length substring of the input. For example, `table[ 0 ][ 0 ]` corresponds to the zero-length substring at the start, `table[ n ][ n ]` corresponds to the zero-length substring at the end, and `table[ 1 ][ 1]` corresponds to the zero-length substring between the first two characters.

The match table's shape is always square, but we only need the elements on or above the diagonal. An element like `table[ 3 ][ 6 ]` represents a substring containing characters at index 3, 4 and 5, but `table[ 6 ][ 3 ]` doesn't represent any substring that's useful to us.

Using a match table for each pattern makes it easy to find all the matches for larger, composite patterns based on the places where their sub-patterns match the input. For composite patterns that contain smaller, sub-patterns, they'll first call the locate() method on their sub-patterns. Then, once a Pattern knows where all its sub-patterns can match parts of the current string, that pattern can determine where it matches parts of the string. For example, imagine we have the pattern `ab+`. It's a composite of the smaller patterns, `a` and `b+` (which is, itself a composite of `b` with a repetition operator). If this this pattern has to find its matches in the input string "baabba", its locate function will ask the `a` pattern to locate all of its occurrences and it will ask `b+` to find all of its occurrences. These two patterns will build the following match tables, showing where they occur in this string.



match table for ‘a’

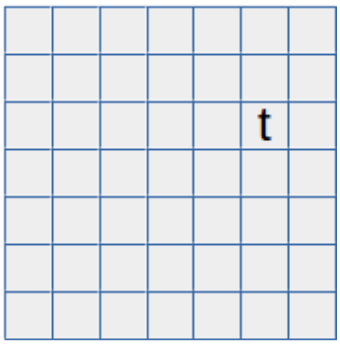


match table for ‘b+’

Match tables for input string "baabba"

In these tables, you can see that there are three matches for the `a` pattern and four matches for the `b+` pattern (three length-1 matches for the three occurrences of 'b', and one more length-2 match for the "bb" substring).

Given these two tables for the matches of its sub-patterns, the locate function for `ab+` can find all places where it matches a substring from "baabba". The `ab+` pattern is just the concatenation of the `a` pattern and the `b+` pattern, so its locate function just needs to find places where there's a match of `a` followed by a match of `b+`. This only happens in two places; `a` has a match from character 2 up to (but not including) 3, and `b+` has a match from character 3 up to (but not including) 5. So, their concatenation has a match from character 2 up to 5. There's also a shorter match that concatenates the same match for `a` with the one-character match of `b+` starting at index 3.



match table for ‘ab+’

Match table for composite pattern on string "baabba"

Completing this project will require implementing pattern subclasses for each part of the regular expression syntax, along with their locate functions. Some of these (like `.`) are easy to implement. Others (like '*' and '+') are a little more difficult. As you complete your

implementation, I'd start with the easy ones, debug your program and check for leaks or other memory errors, then move on to the next one.

Insulation and Simplified Commenting

Notice that the header file for pattern only exposes the constructors for various types of patterns. It doesn't expose the functions that implement their methods or the struct types used to represent them. In fact, these functions are all marked static in the pattern implementation file, so client code couldn't access them if it wanted to. This practice is called *insulation*. Client code can use these functions and the structs used to implement Pattern subclasses, but only via one of their function pointers. It doesn't matter for this project, but insulation would make it possible to change how these subclasses are implemented without breaking or even recompiling client code. This is an important technique if you're writing something like a dynamically linked library, where you may want to change the library version without having to rebuild the applications that use it.

As you can see from the starter, the comments on functions like locateSymbolPattern() and destroyBinaryPattern() are minimal; they just say that these are implementing a particular method for a particular type of pattern. You can continue this practice in your own implementation (for static functions that are just used to implement methods). The thinking here is that the job of these functions is already documented elsewhere. The class they work for should be documented with its struct definition, and the job of the method itself is already documented for you in the definition of the Pattern superclass.

Build Automation

As in previous assignments, you get to create your own Makefile for this assignment. The default rule should build the `regular` target, using separate compile and link steps for building the executable. It should have a clean rule to delete the `regular` executable and any temporary files that could be easily re-created by rebuilding the program. The automated test script depends on your Makefile having a clean rule and a default target builds builds the `regular` executable.

As in recent assignments, include the "-Wall" and "-std=c99" compile flags, along with "-g" to help with debugging.

Testing

Automated Test Script

The starter includes a test script, along with lots of test input files and expected outputs. When we grade your program, we'll test it with this script, along with a few other tests we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
```

```
$ ./test.sh
```

As with previous test scripts, this one reports how it's running your program for each test. This should help you to see how to try out your program on any specific tests you're having trouble with, to figure out what's going wrong.

Testing by Hand

If your program is failing on a test case, you can try out just that one by hand. For a successful test, like test 12, you can run it like:

```
$ ./regular 'ab+c' input-12.txt >| output.txt
$ echo $?
0
$ diff output.txt expected-12.txt
```

Since your program has to use escape sequences to change the text color, diff may not do a good job showing where your output is different from the expected output, especially if it's just a problem with the highlighting. It might be helpful to just take a quick look at your actual program output vs the expected output.

```
# See what my output looks like
$ ./regular 'ab+c' input-12.txt
... program-output ...

# See what the expected output looks like
$ cat expected-12.txt
... expected-output ...
```

If the the problem with your program's output is a problem with whitespace or the ANSI escape sequences used to change the text color, you may want to compare output using hexdump, to see exactly what's in your program's output vs the expected output.

```
# See what my output looks like in hexadecimal
$ ./regular 'ab+c' input-12.txt | hexdump -C
... program-output ...

# See what the expected output looks like.
$ hexdump -C expected-12.txt
... expected-output ...
```

For unsuccessful test cases, your program should exit with a status of 1 and print the particular error message we're expecting. To try out test case 17:

```
$ ./regular '*' input-17.txt 2>| stderr.txt
$ echo $?
1
$ diff stderr.txt stderr-17.txt
```

Testing Extra Credit

We're providing an archive extra6.tgz containing about a dozen additional test cases for the extra credit parts of the assignment. It also has a separate test script, `ectest.sh` to run these tests and report on how they did.

Memory Error and Leaks

On successful test cases, your program is expected to free all of the dynamically allocated memory it allocates and close any files it opens. When your program exits unsuccessfully, this isn't required (since you may be exiting from within a deeply nested sequence of function calls and you may not have access to all resources you've allocated).

Although it's not part of an automated test, we encourage you to try out your executable with valgrind. We certainly will when we're grading your work. Any leaked memory, use of uninitialized memory or access to memory outside the range of an allocated block will cost you some points. Valgrind can help you find these errors before we do.

The compile instructions above include the `-g` flag when building your program. This will help valgrind give more useful reports of where it sees errors. To get valgrind to check for memory errors, including leaks, you can run your program like this:

```
$ valgrind --tool=memcheck --leak-check=full ./regular 'a[bcdef]g' input-09.txt
-lots of valgrind output deleted-
```

With different options, you can get valgrind to check for file leaks instead:

```
$ valgrind --track-fds=yes ./regular 'a[bcdef]g' input-09.txt
-lots of valgrind output deleted-
```

Test Files

The starter includes a test script, `test.sh`, and several input and expected output files to help you make sure your program is behaving correctly. Here's a summary what each of the test cases tries to do:

1. Matching the pattern, `w` on a single-line file that doesn't contain any occurrences of the letter 'w', so the program should not output anything for this case. The starter includes support for literal characters in a pattern, so this
2. Matching the pattern, `h` on a single-line file that has one copy of the letter h, so it should print out the line, with the letter 'h' colored in red. To pass this test, you'll need to add support for printing lines of text that match the pattern, and for highlighting portions of each line that match the pattern.
3. Matching the pattern, `c` against a multi-line file, with four occurrences of the letter c. To pass this test, you'll need to be able to read all the lines from the input file.

4. Matching the pattern, `abc`, a simple test for concatenation. The starter includes support for patterns with concatenation, so this should also be easy to get working.
5. Matching the pattern, `a.c`, a simple test for the '.' pattern, which you get to implement.
6. Matching the pattern, `a..c`. This doesn't test any new types of patterns, but it reads input from stdin, rather than a file given on the command line.
7. Matching the pattern, `^123`, which should only match strings where 123 shows up at the start of a line.
8. Matching the pattern, `wxyz$`, which should only match strings where wxyz shows up at the end of a line.
9. Matching the pattern, `a[bcdef]g`, a simple test of character classes.
10. Matching the pattern, `abc|def|ghi`, a simple test of alternation.
11. Matching the pattern, `ab*c`, a simple test of repetition with '*'.
12. Matching the pattern, `ab+c`, a simple test of repetition with '+'.
13. Matching the pattern, `ab?c`, a simple test of repetition with '?'.
14. Matching the pattern, `a(bc)*d`, a test for using parentheses to control precedence.
15. Matches a larger pattern, '^Your (license|application|program) has been (revoked|accepted|tested)!$'.
16. Uses a pattern that matches real-number constants like 3.14.
17. This unsuccessful test has an invalid pattern, just a `*`.
18. This unsuccessful test has an invalid pattern, with an opening `[` but no closing one.
19. This unsuccessful test tries to read from an input file that doesn't exit.
20. This unsuccessful test runs the program with too many command-line arguments.
21. This unsuccessful test has an input line that's too long.

Grading

We'll be grading your project by making sure it builds cleanly, runs correctly on all our test cases (including some we're not giving out), follows the required design and adheres to the style guide.

- Working makefile: **5 points**

- Program compiles cleanly on the common platform: **10 points**
- The interpreter behaves correctly on all test cases: **80 points**
- Follows the style guide: **20 points**
- Extended syntax for character classes: up to 8 points of **extra credit**
- Extended syntax for repetition: up to 8 points of **extra credit**
- Potential Deductions:
- Up to **-75 percent** for not following the required design.
- Up to **-30 percent** for exhibiting memory or file leaks.
- Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
- **-20 percent** for a late submission.

Getting Started

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on project 2. If you haven't done this yet, go back to the assignment for project 2 and follow the instructions for for cloning your repo.

Unpack the starter into your cloned repo

Make sure you're in the `p6` directory in your cloned repo. You will need to copy and unpack the project 6 starter. We're providing this as a compressed tar archive, starter6.tgz. After you download this file, you can unpack its contents into your `p6` directory. You can do this like you unpacked the starter from previous projects. Be careful; as with the previous projects, this starter contains a `.gitignore`, which will be treated as a hidden file by `ls` and other Unix shell commands. If you unpack elsewhere and then copy into your repo, you may miss this important file. If you `cd` into the `p6` directory of your repo and then unpack, you should get this file and all the other starter files in the right place.

As usual, if you are logged in on one of the common platform systems, you can save yourself a few steps by unpacking the starter directly from our official copy in AFS. Be sure you're in the `p6` directory of your repo and run:

```
$ tar xzvpf /afs/eos.ncsu.edu/courses/csc/csc230/common/www/proj/p6/starter6.tgz
```

Submission Instructions

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files in the p6 directory. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `regular.c` : the main source, mostly written by you.

- `parse.c` : implementation file for the regular expression parser, extended by you
- `parse.h` : header file for the regular expression parser, probably unchanged from the starter.
- `pattern.c` : Object-oriented interface for representing and using regular expressions, extended by you
- `pattern.h` : Header file for the pattern component, extended by you.
- `Makefile` : a Makefile for the project, written by you.
- `input-*.txt` : Input files for testing. The tests look for patterns in these, provided with the starter.
- `expected-*.txt` : expected output to standard output, provied with the starter.
- `stderrr-*.txt` : for the error test cases, this is the expected output to standard error, provied with the starter.
- `test.sh` : automated testing script, provided with the starter.
- `.gitignore` : the new .gitignore file for this project, provided with the starter.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

Whenever you create a new file that needs to go into your repo, you need to stage it for the next commit using the `add` command. You should only need to add each file once. Afterward, you can get git to automatically commit changes to that file:

```
$ git add some-new-file
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've added any new files, you can use a command like the following to commit them, along with any changes to files that were already being tracked:

```
$ git commit -am "<meaningful message for future self>"
```

Remember, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ unset SSH_ASKPASS # if needed
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to the previous projects. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for project 6. This job polls

your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules.
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and identify errors that occur during each phase. In particular, they will be able to describe the differences in this process between C and Java.
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow.
- Explain, inspect, and implement programs using structures such as enumerated types, unions, and constants and arithmetic, logical, relational, assignment, and bitwise operators.
- Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O.
- Use simple command-line tools to design, document, debug, and maintain their programs.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or Git, to track changes and do parallel development of software.

- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.