CSC173: Project 2
Grammars and Parsing

Requirements

1. Implement a recursive-descent parser for a context-free grammar of regular expressions.

   - Your parser should read successive expressions from standard input, construct a parse tree for the expression if it is well-formed, and then print that parse tree to standard output (see description below).
   - If the input is not well-formed, your parser should print an appropriate message and resume processing at the next line of input (*i.e.*, skip to the next newline).
   - You may assume that the regular expressions are intended to match strings using only the lowercase letters `a` through `z`. The regular expressions themselves may include certain additional symbols, of course.
   - You may assume that expressions are on a single line of input.
   - Your parser should terminate on end-of-file on standard input.

2. Implement a table-driven parser for the grammar of regular expressions.

   - As for Part 1: read expressions from standard input, try to parse the input, print the parse tree or an error message.
   - Most of the infrastructure of the parser will be the same as for Part 1.
   - You must use an explicit parsing table that references explicitly-represented productions (FOCS Figs. 11.31 and 11.32). You should have a function that creates and returns an instance of this table for your grammar of regular expressions.
   - You must have a single parsing function that uses a parsing table to perform a table-driven parse. (You may have helper functions also.)
   - It may be helpful to produce output like FOCS Fig. 11.34 during debugging.

Part 3 is on the next page. . .

3. The final step is to take the parse tree built by your parser for a regular expression and construct an equivalent finite automaton. This is sometimes called "compiling" the regular expression. This is not easy, and you don't have to do all of it to get full points on the project. So read the requirements for this part carefully.

   First, you need to be able to interpret the parse tree as an *expression tree* (FOCS Section 5.2, pp. 228–230, and elsewhere in Chapter 5). This is complicated by the form of the grammar that you must use with recursive descent parsing (see below). But once you have an expression tree, you can easily print it as a nested set of prefix expressions (an example is given in an appendix).

   That is all you need to do for this part of the project. You may use either version of your parser—they should produce equivalent parse trees.

   But if you want to make it all come together. . .

   Once you have an expression tree, you would create from it an equivalent NFA-ϵ following the procedure seen in Unit 1 (but not in Project 1). You may need to extend your representation of NFAs from Project 1 in order to support ϵ-tranmsitions.

   You then need to implement the ϵ-elimination algorithm to produce a plain NFA. This is not conceptually hard, but there is quite a lot to keep track of. You might try doing it in Java before jumping into the C.

   Once you have an NFA, you can execute it or, better, convert it to a DFA using Part 3 of your Project 1. From there, you can run the DFA on input strings to test whether they match the original regular expression.

   The result is a command-line pattern matcher like the UNIX program `grep`.

You must implement all three parts of the project as a single program named `rexp`. This program should read the input, call both parsers and the "compiler" (if you do all three parts), and print the results informatively. It is your responsibility to make it clear to us what your program is doing.

Note: You can, of course, implement the parts one at a time. That's probably a good strategy. In your main loop, only call the parts that you have implemented so far.

FYI: *The UNIX Programming Environment* by Kernighan and Pike, Chapter 8, describes building a similar program (for arithmetic expressions) using a compiler-generator (`yacc`, now superceded by `bison`). Of course you're writing the parser by hand, but some of the ideas might be of interest, if not immediately useful.

There is no opportunity for extra credit in this project.

Grammars for Regular Expressions

Here is a simple context-free grammar for regular expressions matching strings over `a...z`, based on their recursive definition:

$$\begin{aligned}
\langle E \rangle &\to \langle E \rangle \ \texttt{|} \ \langle E \rangle \\
\langle E \rangle &\to \langle E \rangle \langle E \rangle \\
\langle E \rangle &\to \langle E \rangle \star \\
\langle E \rangle &\to \ \texttt{(} \ \langle E \rangle \ \texttt{)} \\
\langle E \rangle &\to \ \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z}
\end{aligned}$$

Note that the symbol "`|`" in the first production is the symbol for the union operator in the language of regular expressions, not the meta-character "|" used in context-free grammars.

Ask yourself if this grammar is parsable by a recursive descent parser. You'd want to be able to figure this out, so give it a try.

If it is parsable, then you are all set. If not, think about what you have to do to make it parsable. See FOCS p. 624 and the section "Making Grammars Parsable" pp. 631–633, and think about parsing an operator that is not marked with a symbol in the input.

An appendix at the end of this document has more help, but if you want to learn something, **don't read it until until you have tried to figure this out for yourself.**

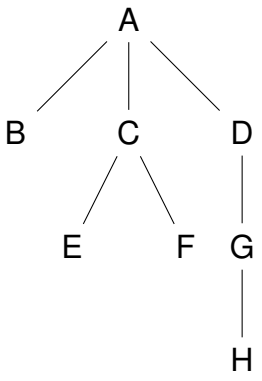
Parse Trees and Printing Parse Trees

A parse tree is a dynamic data structure. You have seen trees in Java and they're the same in C. The textbook has an entire chapter on trees (Chapter 5), and if you read the chapter for this unit I promise you that you will find all kinds of useful code.

For parts 1 and 2 of this project, your program must print the resulting parse tree to standard output. There are many ways to do this, but for this project you will produce output in an *indented, pretty-printed format*. What this means is:

- Each node is printed on a separate line.
- The children of a node are indented relative to their parent.
- All children of a node are at the same level of indentation.

Here is an example:




```
A
   B
   C
      E
      F
   D
      G
         H
```

Printing a tree involves doing a tree traversal, right? What kind of traversal? Think about it. . . Traversing a tree is a recursive procedure, right? You print nodes and you print their children, in the right order. So you can implement it using a recursive function. It's elegant *and* practical.

You also need to keep track of the current indentation level. So this will be a parameter to your printing function. In C, which does not have function overloading, this usually means two functions: a toplevel pretty-print function with no indentation parameter, and an internal function with that parameter, called from the toplevel function with indentation 0 to get the ball rolling.

Project Submission

Your project submission MUST include the following:

1. A `README.txt` file or `README.pdf` document describing:
   (a) Any collaborators (see below)
   (b) How to build your project
   (c) How to run your project's program(s) to demonstrate that it/they meet the requirements
2. All source code for your project. Eclipse projects must include the project settings from the project folder (`.project`, `.cproject`, and `.settings`). Non-Eclipse projects must include a `Makefile` or shell script that will build the program per your instructions, or at least have those instructions in your README.txt.
3. A completed copy of the submission form posted with the project description. Projects without this will receive a grade of 0. If you cannot complete and save a PDF form, submit a text file containing the questions and your (brief) answers.

We must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better grade your will be.** It is your job to make both the building and the running of programs easy and informative for your users.

Programming Policies

You must write your programs using the "C99" dialect of C. This means using the "`-std=c99`" option with `gcc` or `clang`. For more information, see Wikipedia.

You must also use the options "`-Wall -Werror`". These cause the compiler to report all warnings, and to make any warnings into errors that prevent your program from compiling. You should be able to write code without warnings in this course.

With these settings, your program should compile and run consistently on any platform. We will deal with any platform-specific discrepancies as they arise.

If you submit an Eclipse project, it must have these settings associated with the project. Projects with that compile with warnings will be considered incomplete.

Furthermore, your program should pass `valgrind` with no error messages. If you don't know what this means or why it is A Good Thing, look at the C for Java Programmers document which has a short section about it. Programs that do not receive a clean report from `valgrind` have problems that **should be fixed** whether or not they run properly. If you are developing on Windows, you will need to look for alternative memory-checking tools.

Late Policy

Late projects will **not** be accepted. Submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

Collaboration Policy

You will learn the most if you do the projects YOURSELF.

That said, collaboration on projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC173.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the group should submit on the group's behalf and the grade will be shared with other members of the group. Other group members should submit a short comment naming the other collaborators.
- All members of a collaborative group will get the same grade on the project.

Academic Honesty

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

There is code out there for all these projects. You know it. We know it.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.

Appendix: R-D-Parsable Grammar of Regular Expressions

You should try to figure this out for yourself before reading this section.

Really: Go think about it. You should know how to make a grammar parseable by a recursive-descent parser. I will say that handling the closure operator takes some thinking (at least, it took me some thinking).

Then go to the next page if you still can't figure it out by yourself.

Original grammar of regular expressions based on their recursive definition:

$$\begin{aligned}
\langle E \rangle &\to \langle E \rangle \texttt{|} \langle E \rangle \\
\langle E \rangle &\to \langle E \rangle \langle E \rangle \\
\langle E \rangle &\to \langle E \rangle \texttt{*} \\
\langle E \rangle &\to \texttt{(} \langle E \rangle \texttt{)} \\
\langle E \rangle &\to \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z}
\end{aligned}$$

Equivalent unambiguous, not left-recursive, left-factored grammar that enforces the correct operator precedence and is parsable by a recursive-descent parser:

$$\begin{aligned}
\langle E \rangle &\to \langle C \rangle \langle ET \rangle \\
\langle ET \rangle &\to \texttt{|} \langle E \rangle \mid \epsilon \\
\langle C \rangle &\to \langle S \rangle \langle CT \rangle \\
\langle CT \rangle &\to \texttt{.} \langle C \rangle \mid \epsilon \\
\langle S \rangle &\to \langle A \rangle \langle ST \rangle \\
\langle ST \rangle &\to \texttt{*} \langle ST \rangle \mid \epsilon \\
\langle A \rangle &\to \texttt{(} \langle E \rangle \texttt{)} \mid \langle X \rangle \\
\langle X \rangle &\to \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z}
\end{aligned}$$

The syntactic categories are named E for expression, C for concatenation, S for "star" (closure), A for atomic expression, and X for input symbol. The T versions of the categories are the "tail" productions created by left-factoring the grammar per FOCS Ex. 11.15.

Note the addition of an explicit dot ("`.`") for the concatenation operator. That's ok. Just remember to include it in the inputs to your parser.

Evaluating the expressions produced by this grammar is a bit more involved than the nice almost-expression trees produced by the original grammar. But you should be able to walk the tree and evaluate it anyway. Try a couple of expressions, look at the parse trees, and figure out how to do it.

Here's a hint: the "non-T" productions always parse something, and then parse their corresponding "T" version. The "T" version always does something with what was parsed originally. So you might want to pass that first value in to the function that evaluates the T version, so that it can do its thing (which depends on which T production it was). ST is interesting.

Take your questions to study session but don't expect the TAs to solve the problem for you because that's not their job.

Appendix: Sample Output

Here is example output from my program for the input "`a|b.c*`".

```
E
  C
    S
      A
        X
          a
      ST
        eps
    CT
      eps
  ET
    |
    E
      C
        S
          A
            X
              b
          ST
            eps
        CT
          .
          C
            S
              A
                X
                  c
              ST
                *
                ST
                  eps
            CT
              eps
      ET
        eps
```

And here is the regular expression as an expression tree printed in prefix notation:

```
(UNION (ATOMIC a) (CONCAT (ATOMIC b) (CLOSURE (ATOMIC c))))
```

From this you could construct the equivalent automaton and run it to match strings against the original regular expression.