Project 1: Recursive-Descent Parser

Due Date: 9/21 by 11:59p

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

Aims of This Project

The aims of this project are as follows:

- To familiarize you with recursive-descent parsing.
- To expose you to Java programming.
- To provide familiarity with the tools used to build programs on the `remote.cs` Linux systems.

Project Specification

A **ugly-regexp** is defined inductively as follows:

- A *primitive* ugly-regexp is `chars(`*x_1*`,` ... `,`*x_n*`)` with *n* `>= 1`. The `chars` and the comma `,` occur literally, whereas the *x_1*, ..., *x_n* represent any single character. A `chars` ugly-regexp represents the character class containing the characters *x_1*, ..., *x_n*.
- If *U1* and *U2* are ugly-regexp's, then so are:
  - `(` *U1* `)` representing the same ugly-regexp as *U1*.
  - `*` *U1* representing the Kleene closure of ugly-regexp *U1*.
  - *U1* + *U2* representing the alternation of ugly-regexp's *U1* or *U2*.
  - *U1* `.` *U2* representing the concatenation of ugly-regexp's *U1* and *U2*.

The operators precedences are strictly ordered `.` (lowest), +, `*` (highest). The `.` and + binary operators are left-associative. The `*` prefix operator is allowed to nest, i.e., `**`*U1* is legal. Parentheses are used in the usual way to override the default associativity and precedence.

An ugly-regexp is terminated by a newline. Other linear-whitespace (spaces or tabs) should be ignored. It is assumed that an ugly-regexp contains only ASCII characters.

Examples of legal ugly-regexp's, followed by a #-comment containing the equivalent regexp in ″standard″ syntax:

```
chars(a, b)                                  #[ab]
chars(,, ))                                  #[\,\)]
chars(a, b) + chars(1, 2) . chars(3, 4)      #([ab]|[12])[34]
chars(a, b) + **(chars(1, 2) . chars(3, 4)) #[ab]|([12][34])**
```

[**Note**: the `#`-comment comment above is **not** the translation your program is required to produce.]

An ugly-regexp *U* can be translated into standard regexp syntax using a translation function *T(U)* which is defined as follows:

- If *U* is of the form `chars(`*x_1*`, ...,`*x_n*`)`, then *T(U)* is `[`*x_1...x_n*`]`. If any *x_i* is not a letter or a digit, then it must be escaped by preceeding it with a single backslash character `\`.
- If *U* is of the form `(` *U1* `)`, then *T(U)* is `(`*T(U1)*`)`.
- If *U* is of the form `*` *U1*, then *T(U)* is *T(U1)*`*`.
- If *U* is of the form *U1* + *U2*, then *T(U)* is `(`*T(U1)*`|`*T(U2)*`)`.
- If *U* is of the form *U1* `.` *U2*, then *T(U)* is `(`*T(U1)T(U2)*`)`.

You are required to write a java8 program which will parse a ugly-regexp and translate it into standard regexp syntax. Specifically, submit a `prj1.tar.gz` archive, which when unpacked into an empty directory on the `remote.cs` machines, will build a `target/prj1.jar` jar-file when given the command `make`. When the generated jar-file is run using the command:

```
$ java -cp target/prj1.jar edu.binghamton.cs571.UglyRegexpParser
FILENAME
```

will read ugly-regexp's from *FILENAME* (one-per-line) and output the translation of each ugly-regexp on standard output followed by a newline. The program should read from standard input if *FILENAME* is specified as a single dash `-`.

Additionally, the submitted `prj1.tar.gz` archive **must** contain a `README` file which should minimally contain your name, B-number and email. Additionally, it may contain the status of your project and any other information you believe is relevant.

Rationale for the Requirements

This is basically a ″make-work″ project in that a ugly-regexp is not particularly useful. The rationales for the idiosyncratic syntax included the following:

- The relative precedence of the alternation and concatentation operators in ugly-regexp's differs from that in standard regexp syntax. This makes it impossible to do the translation using simple string replacement and requires writing a full parser.
- Similar considerations led to the use of a prefix rather than suffix Kleene-closure operator `*`.

- The translation introduces possibly redundant parentheses (in the translations for parentheses, alternation, concatenation). This simplifies the translation function considerably.
- Since whitespace is ignored, the ugly-regexp syntax does not allow any kind of whitespace characters within the constructed regexp's. This simplifies the scanner slightly.

Example Log

Here is an annotated log of an interactive session with the program:

```
$ java -cp target/prj1.jar edu.binghamton.cs571.UglyRegexpParser -
chars(a, b)
[ab]
* chars(,,)
<stdin>:2:11: syntax error at '      #syntax error at newline with newline
', expecting ')'                     #quoted within single-quotes
* chars(,)
[\,]*
chars(a) + chars(b) . chars(c)       #+ binds tighter than .
(([a]|[b])[c])
chars(a) + (chars(b) . chars(c))
([a]|([b][c]))                       #double paren due to paren & . translation
chars(a) + (chars(b) . *chars(c))
([a]|([b][c]*))
chars(a) + * * (chars(b) . chars(c)) # *'s nest
([a]|([b][c]))**)
(chars(a,))                          #) after comma treated as chars char
<stdin>:8:11: syntax error at '
', expecting ')'
(chars(a,)))                         #fixed here
([a\)])
$
```

Note that EOF is signalled to the Unix terminal-controller by typing a control-D character.

Provided Files

The ./files directory contains the following:

Makefile
: This file assumes that the project is set up to be built using a ant `build.xml` build-file. Simply typing `make` will build the project, `make clean` will remove all generated artifacts and `make submit` will create a `prj1.tar.gz` archive containing the files to be submitted.

  You may edit this file if you choose to use a different organization for your project. When editing, watch out for tabs (the first character of any command-line **must be a tab character**).

An ant buildfile
: This ant build-file compiles all source files from the `src` directory. It generates `.class` files in the `build` directory and `.jar` files in the `target` directory.

README
: A template README; replace the `XXX` with your name, B-number and email. You may add any other information you believe is relevant to your project submission. In particular, you should document the data-structure used for your word-store.

`src`
: Source files which contain all the code necessary for the project except for the guts of the regular expression parsing. Specifically it provides the following java classes (ordered from the bottom-up):

  Coords
  : Essentially a `struct` which tracks the source position (filename, line-number, column-number) of a token.

  Token
  : Essentially a struct which defines a token. The implementation file also contains a `Token.Kind enum` which defines all the different types of tokens for this project.

  Scanner
  : A very crude scanner which delivers tokens of kind `Token.Kind` while ignoring linear whitespace.

  UglyRegexpParser
  : A skeleton file which will need to be completed by you. It contains the necessary `main()` function for your program as well as all error handling and utility functions for parsing and translation.

Hints

You may choose to follow the following hints (they are not by any means required). They assume that you are using the project structure supported by the provided Makefile, and ant buildfile.

Note that ugly-regexp's treat characters like `(`, `)`, `,`, `.` and + in two different ways:

1. As meta-characters. For example, `.` is used as a meta-character denoting concatenation.

2. As regular characters when used within `chars()`. For example, (the first) `,`, `.` and + are not meta-characters in `chars(,, ., +)` (the comma's other than the first are meta-characters).

A situation like this where the kind of a token depends on the context is handled usually in 2 ways:

1. The scanner recognizes such characters as special and delivers them to the parser as special tokens. Then if the parser sees such a special token being delivered in a context where it should be treated as an ordinary character, it makes adjustments to treat it as a ordinary character.

2. The scanner treats such characters simply as regular characters. Then if the parser sees such a character being delivered in a context where it should be treated as a special character, it makes adjustments to treat it as a special character.

The provided scanner uses (2). The `check()` and `match()` utility functions provided in the skeleton parser allow an optional `lexeme` parameter which permits using the lexeme to guide parsing decisions.

You may proceed as follows:

1. Review material on regexp's, cfg's and recursive-descent parsers. Make sure you understand the following:

   - The standard syntax of regexp's.
   - How to build a grammar to enforce specific associativity and precedence of operators.
   - How to transform a grammar to remove left-recursive rules.
   - How to construct a recursive-descent parser given a grammar amenable to recursive-descent parsing.

2. Construct a grammar for ugly-regexp's which enforces the required associativity and precedence. You may use the grammars given in the slides as guides; specifically, the grammars for comma-separated `ID`'s and arithmetic expressions may prove to be useful.

3. Transform the above grammar to remove left-recursive rules and make it amenable to recursive-descent parsing.

4. Add parsing functions to the `UglyRegexpParser` ignoring translation. Your aim at this step should be to build a recognizer: i.e. if the provided input is syntactically legal then the program should terminate silently; OTOH, if the input is not syntactically legal, the program should output a suitable error message. Since you are ignoring translation, set up all the parsing functions to simply return `void` or a `null` string.

   It may be best to do this from the bottom-up. First get `chars()` working, then `*`, `()`, followed by `+` and `.`.

5. Add translation logic. Make each parsing function return a string which corresponds to the translation of the input recognized by that function. Some of the parsing functions will need input parameters representing translations of input seen earlier.

   You may use the provided `quote()` function to quote characters within a character class as per the project requirements.

   [Note: Standard Java practice suggests using mutable structures like `StringBuffer` or `StringBuilder` to concatenate strings; it may be difficult to do so in this program and you should simply use string concatenation using +).

6. Test and review your code until it meets all requirements.

Submission

You will need to submit a compressed archive file `prj1.tar.gz` which contains all the files necessary to build your jar file. Additionally, this archive **must** contain a `README` file which should minimally contain your name, email, the status of your project and any other information you believe is relevant.

If you are using the suggested project structure, then the provided Makefile provides a `submit` target which will build the compressed archive for you; simply type `make submit`.

Note that it is your responsibility to ensure that your submission is complete so that simply typing `make` builds the jar file. To test whether your archive is complete, simply unpack it into a empty directory and see if it builds and runs correctly.

To submit the above archive, please use blackboard by following the `Content->Projects` link.