CS 452/552: Assignment 3

Due: December 1, 2023, by 11:00 PM (Central)

Overview

Your goal for this assignment is to write a program that functions as a knowledge base which can track information expressed in propositional logic and answer queries through the resolution refutation inference procedure. Grading notes can be found on page 7.

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "resolution in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must **not** make use of any non-standard or third-party libraries. In particular, you cannot use such libraries to help parse the string representations of sentences in propositional logic.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**

Program Requirements

Your program should be runnable from the command line, and it should be able to process command-line arguments to update various program options and parameters as needed. There is one optional command-line argument which is listed below:

- `<FILENAME>`: Reads and processes a set of commands (one per line) from the text file named `<FILENAME>` (specified as a `String`); the filename may include path information, so do not assume that the file is located in the same directory as your source code. Commands are described on page 3.

Because there is only one possible command-line argument, it is not necessary to use flags (e.g., `-f`) to differentiate between arguments. If no filename is provided, then the program should run in **interactive mode**.

For example, if you write a Java program with `main` method in a class named `KBDriver`, then running your program as shown below would start **interactive mode**:

```
shell$ java KBDriver
Welcome to the Knowledge Base!
Please TELL or ASK me anything!
(type HELP for more information)
>
```

The `>` prompt indicates that the program is waiting for the user to enter a command, which the program will read from standard input (`System.in` in Java). After the user types a command and presses the `Enter` key on the keyboard, the program should read the user's input and process the request appropriately.

An example of an interactive session is shown below:

```
shell$ java KBDriver
Welcome to the Knowledge Base!
Please TELL or ASK me anything!
(type HELP for more information)
> TELLC P v Q
> TELLC ~Q
> ASK P
Yes, KB entails P
> DONE
Thank you for using the Knowledge Base!
```

If the program is run with a filename specified via a command-line argument, then the program should read commands from the file one line at a time and print each command along with any associated output. Blank lines and lines beginning with a `#` character should be ignored. As an example, the `ex2a-ask-literal.txt` file contains the three commands shown below:

```
# Basic example of the elimination rule
TELLC P v Q
TELLC ~Q

ASK P
```

The program should produce the following output when run with this file:

```
shell$ java KBDriver ex2a-ask-literal.txt
> TELLC P v Q
> TELLC ~Q
> ASK P
Yes, KB entails P
```

Note that the program should be able to detect the end of the file and terminate even if the file does not include a `DONE` command at the end. Any program that does not work this way will lose points from the assignment grade. In particular, your program should not require that the source code be edited (and possibly re-compiled) in order to run with different inputs.

Commands for the Knowledge Base

The following commands should be supported by your knowledge base program. The command names are listed below in all uppercase letters to make them stand out, but your program should be able to handle these commands regardless of casing (e.g., `HELP`, `help`, and `hElP` should all be recognized as the `HELP` command).

- `HELP`
  Prints a list of supported commands along with brief descriptions of their behavior.
- `DONE`, `EXIT`, `QUIT`
  Terminates the session.
- `TELLC <clause>`
  Adds the given `<clause>` to the knowledge base. See the next section for details on how to represent a `<clause>`.
- `PRINT`
  Prints the clauses currently in the knowledge base.
- `ASK <query>`
  Determines if the knowledge base entails `<query>` using the **resolution refutation method** (discussed in lecture 05-4). The `<query>` can be either a single literal or an arbitrary sentence in propositional logic. (To get started, your program should just support literals as queries; once you get that working, you can add support for arbitrary sentences.)
- `PROOF <query>`
  Prints a proof of `<query>` from the knowledge base, obtained via the **resolution refutation method**. The last line in the proof should be the empty clause, denoted with `()`, and the preceding lines should be the clauses that are resolved to reach it.
- `PARSE <sentence>`
  Prints the parse tree of the given `<sentence>` in (simplified) propositional logic.
- `CNF <sentence>`
  Prints the **conjunctive normal form** representation of the given `<sentence>` in (simplified) propositional logic.
- `TELL <sentence>`
  Adds the clauses in the CNF representation of `<sentence>` to the knowledge base.

The representations of `<clause>`, `<query>`, and `<sentence>` values are explained in the next section, followed by example output from these commands.

Representing Propositional Logic in Plain Text

The following rules will be used for writing propositional logic sentences in plain text:

- `(` and `)` are used for grouping expressions with parentheses
- `~` (the tilde character) is used for the $\neg$ connective (logical negation)
- `^` (the caret character) is used for the $\wedge$ connective (logical and)
- `v` (a lowercase 'v') is used for the $\vee$ connective (logical or)
- `=>` is used for the $\Rightarrow$ connective (conditional)
- `<=>` is used for the $\Leftrightarrow$ connective (biconditional)
- Alphanumeric sequences are used for proposition symbols, with the restriction that the letters 'v' (`v`) and 'V' (`V`) **cannot** be used as part of a symbol (the former restriction is because `v` is used for $\vee$, while the latter restriction is to avoid confusion). Symbol names are **case-sensitive**, so `p` and `P` represent **different** proposition symbols.
- White space should be **ignored**.

Recall that a **clause** is a disjunction of **literals**, with each **literal** being a proposition symbol or a negated proposition symbol. The string representation of a clause includes only `~`, `v`, and alphanumeric sequences for proposition symbols (ignoring spaces). For example, several string representations are shown below, with the corresponding clauses on the right:

```
P v Q v ~R
~B11 v P12 v P21
P v ~Q v R v S v ~T
svtvu
This is a long literal v A short literal
```

$$P \vee Q \vee \neg R$$
$$\neg B_{11} \vee P_{12} \vee P_{21}$$
$$P \vee \neg Q \vee R \vee S \vee \neg T$$
$$s \vee t \vee u$$
$$Thisisalongliteral \vee Ashortliteral$$

In particular, the last two examples above show that whitespace within the string representation of a clause is **ignored** (even if it might help improve readability). So both `foo bar` and `foobar` refer to the same literal *foobar*. To avoid confusion, most proposition symbols should be written in plain text without spaces (e.g., `foobar` instead of `foo bar`). Additionally, to improve readability, space should be used to separate connectives from symbols (e.g., `p v q` instead of `pvq`).

A **sentence** in propositional logic can use **any** of the logical connectives along with parentheses to represent arbitrarily complex expressions. However, to **simplify** things, we will restrict our attention to sentences that follow the **BNF grammar** below:

$$\begin{aligned}
Sentence &\rightarrow UnarySentence \mid BinarySentence \\
UnarySentence &\rightarrow Symbol \mid (Sentence) \mid \neg UnarySentence \\
BinarySentence &\rightarrow UnarySentence \wedge UnarySentence \\
&\mid\ \ UnarySentence \vee UnarySentence \\
&\mid\ \ UnarySentence \Rightarrow UnarySentence \\
&\mid\ \ UnarySentence \Leftrightarrow UnarySentence \\
Symbol &\rightarrow P \mid Q \mid R \mid \ldots
\end{aligned}$$

In particular, this grammar **requires** that complex expressions involving multiple binary connectives be parenthesized to capture precedence. For example, the expression $P \wedge Q \Rightarrow R$ is unambiguous in propositional logic because $\wedge$ has precedence over $\Rightarrow$. However, the above grammar **requires** that this expression be written in plain text as `(P ^ Q) => R`. This is primarily to help identify the **outermost** binary connective in a complex expression (assuming that there is one).

Below are several string representations, along with the corresponding sentences on the right:

```
(P ^ Q) => R
~(P v (Q <=> ~R))
P v (Q v (R v (S v T)))
~~~(P v ~~~Q)
~~(~(~~(P v Q)))
```

$$(P \wedge Q) \Rightarrow R$$
$$\neg(P \vee (Q \Leftrightarrow \neg R))$$
$$P \vee (Q \vee (R \vee (S \vee T)))$$
$$\neg\neg\neg(P \vee \neg\neg\neg Q)$$
$$\neg\neg(\neg(\neg\neg(P \vee Q)))$$

Recall that our representation of clauses does **not** allow for the use of parentheses, which means that any clause with more than two literals will **not** fit this grammar (e.g., $P \vee Q \vee R$ is not a sentence in this grammar, but $P \vee (Q \vee R)$ is). This means that your program **should probably** handle **clauses** and general **sentences** in (simplified) propositional logic **differently**! (*Hint:* Clauses should be relatively easy to read from strings, but general sentences require a bit more work! This is why there are separate `TELLC` and `TELL` commands.)

You can assume that your program will only be run with valid string representations of clauses and sentences, so it is not necessary to do error-checking or identify malformed expressions.

Output

Below are several interactive runs that show the basic behavior of various commands that should be supported. Your program's output should be similar but does not need to match exactly.

```
shell$ java KBDriver
Welcome to the Knowledge Base!
Please TELL or ASK me anything!
(type HELP for more information)
> TELLC P v Q
> PRINT
(P v Q)
> TELLC ~Q
> PRINT
(P v Q)
(~Q)
> ASK P
Yes, KB entails P
> PROOF P
Proof:
 1. P v Q                  [Premise]
 2. ~Q                     [Premise]
 3. ~P                     [Negated Goal]
 4. Q                      [Resolution on P: 1, 3]
 5. ()                     [Resolution on Q: 2, 4]
> ASK Q
No, KB does not entail Q
> PROOF Q
```

```
No proof exists
> DONE
Thank you for using the Knowledge Base!
```

Examples of parsing:

```
shell$ java KBDriver
Welcome to the Knowledge Base!
Please TELL or ASK me anything!
(type HELP for more information)
> PARSE P => (Q ^ R)
Orig: [P=>(Q^R)] Binary [=>]
 LHS: [P] Unary [symbol]: [P]
 RHS: [(Q^R)] Unary [()]
  Sub: [Q^R] Binary [^]
   LHS: [Q] Unary [symbol]: [Q]
   RHS: [R] Unary [symbol]: [R]
> PARSE P v (Q v (R <=> ~S))
Orig: [Pv(Qv(R<=>~S))] Binary [v]
 LHS: [P] Unary [symbol]: [P]
 RHS: [(Qv(R<=>~S))] Unary [()]
  Sub: [Qv(R<=>~S)] Binary [v]
   LHS: [Q] Unary [symbol]: [Q]
   RHS: [(R<=>~S)] Unary [()]
    Sub: [R<=>~S] Binary [<=>]
     LHS: [R] Unary [symbol]: [R]
     RHS: [~S] Unary [~]
      Sub: [S] Unary [symbol]: [S]
> PARSE (P <=> ~Q) ^ (~R => S)
Orig: [(P<=>~Q)^(~R=>S)] Binary [^]
 LHS: [(P<=>~Q)] Unary [()]
  Sub: [P<=>~Q] Binary [<=>]
   LHS: [P] Unary [symbol]: [P]
   RHS: [~Q] Unary [~]
    Sub: [Q] Unary [symbol]: [Q]
 RHS: [(~R=>S)] Unary [()]
  Sub: [~R=>S] Binary [=>]
   LHS: [~R] Unary [~]
    Sub: [R] Unary [symbol]: [R]
   RHS: [S] Unary [symbol]: [S]
```

Examples of CNF conversion:

```
shell$ java KBDriver
Welcome to the Knowledge Base!
Please TELL or ASK me anything!
(type HELP for more information)
> CNF P => Q
Result: (~P v Q)
> CNF ~(P^Q)
Result: (~P v ~Q)
> CNF (A <=> B)
Result: (~A v B) ^ (A v ~B)
```

Examples of `TELL` and `ASK` with general sentences:

```
shell$ java KBDriver
Welcome to the Knowledge Base!
Please TELL or ASK me anything!
(type HELP for more information)
> TELL P => Q
> TELL ~Q
> ASK ~P
Yes, KB entails ~P
> TELL B11 <=> (P12 v P21)
> TELL ~B11
> ASK P12 v P21
No, KB does not entail P12 v P21
> ASK ~P12 ^ ~P21
Yes, KB entails ~P12 ^ ~P21
> QUIT
Thank you for using the Knowledge Base!
```

Additional files for testing and example output can be found on Canvas.

Grading

At a high level, your assignment grade will depend on the following:

1. Support for basic commands `HELP`, `DONE`, `TELLC`, and `PRINT`
2. Support for `ASK` with **literals** as queries
3. Support for `PROOF` with **literals** as queries
4. Support for `PARSE` for general **sentences**
5. Support for `CNF` for general **sentences**
6. Support for `TELL` with general **sentences**
7. Support for `ASK` and `PROOF` with general **sentences**

In order to earn points in one of these categories, your program needs to be correct or nearly correct across all preceding categories. For example, a program that implements `TELLC` **incorrectly** cannot earn full credit for `ASK` with literals because `ASK` requires that the knowledge base be populated with the correct clauses in order to function properly. This means that you should focus your implementation efforts on these categories **in order**, and that you should be reasonably sure that each category is working properly **before** moving on to the next one. Simply throwing together a bunch of partially correct code for each of the categories is likely to result in a **bad grade**!