Exercise 2 - Resolution in propositional logic

- As the name of the function we have to implement hints at us, a recursive procedure is required. The procedure itself is not far from intuitive. Simply put, given a clause, and a resolved knowledgebase, we simply have to backtrace what resolution lead to that resolvent. We can iterate trough the knowledge base to find both resolvers. A thing to look out for, is that there may be multiple resolutions with different resolvers that lead to the same conclusion. Since we know every clause is either a premise, or a conclusing from premises (and earlier conclusions), we can stop at the first encounter. When this happens, we can try and find what both resolvers were conclusions of using the same procedure, so we recurse. Note that if no resolvers were found, we must have stumbled upon a premise, and therefore do not recurse.
  In `C`, we can use some of the given functions to check each resolvent. Furthermore, we added function to check whether the given conclusing is in a clause set, and another to print the resolution.

- Instead of fully replacing the `init()` function, we decided to add a new function `loadKB()`, and use `init()` as a backup. You can run either one using the following syntax: `./a.out <mode:(1,2,3)> <string>`, where you select a mode for each operation. 1 is `init()`, 2 is `loadKB()`, and 3 is `loadKBfromString()`, which is from the bonus exercise. As an example, the given input for our program would be:

  ```
  ./a.out 2 KB=[[~a,~b],[a,~b,~c,~d],[b,~d],[c,~d],[d]]
  ```

  Since most of the foundation for parsing the actual string was already in place, we only had to split the string in analysable substrings for the parser to process. With the help of the built in function `strncpy`, this was easily managed.
  A thing we paid special attention to, was error checking. Since long strings can be manually entered as a parameter, a small mistake could be easily glossed over. Therefore, we added a minimal, yet specific error check in each analysis we perform, and report to the user when he/she has made a syntax mistake.

- After a little bit of puzzling, we came up with the following string

  ```
  KB=[[a],[~m],[~a,b],[~b,c],[~c,~d],[~b,d,~e],
  [d,e,f],[e,~f,g],[~g,~h],[h,l],[~g,m]]
  ```

  This produces the following proof

```
[e,~f,m] is inferred from [e,~f,g] and [~g,m].
[~b,d,~f,m] is inferred from [~b,d,~e] and [e,~f,m].
[~b,~c,~f,m] is inferred from [~c,~d] and [~b,d,~f,m].
[~b,~c,d,e,m] is inferred from [d,e,f] and [~b,~c,~f,m].
[~a,~c,d,e,m] is inferred from [~a,b] and [~b,~c,d,e,m].
[~a,~b,~c,d,m] is inferred from [~b,d,~e] and [~a,~c,d,e,m].
[~b,~c,d,m] is inferred from [a] and [~a,~b,~c,d,m].
[~a,~c,d,m] is inferred from [~a,b] and [~b,~c,d,m].
[~a,~b,d,m] is inferred from [~b,c] and [~a,~c,d,m].
[~a,~b,~c,m] is inferred from [~c,~d] and [~a,~b,d,m].
[~b,~c,m] is inferred from [a] and [~a,~b,~c,m].
[~a,~c,m] is inferred from [~a,b] and [~b,~c,m].
[~a,~b,m] is inferred from [~b,c] and [~a,~c,m].
[~b,m] is inferred from [a] and [~a,~b,m].
[~a,m] is inferred from [~a,b] and [~b,m].
[~a] is inferred from [~m] and [~a,m].
[]=FALSE is inferred from [a] and [~a].
```

What struck us, was that we intended this proof to be (about) 20 steps, and solving by resolving $m \wedge \neg m \Rightarrow \emptyset$. The program however, found a quicker (17 steps) solution by resolving $a \wedge \neg a \Rightarrow \emptyset$. This shows that there were in fact more ways to proof this sentence.

- To complete this exercise, we created an entirely sepperate mode for the program, namely 3, which can also accept any expression of any form, and including the implication and bi-implication. To give an example, the expression

$$(a \Rightarrow b) \vee \neg(c \Leftrightarrow (d \wedge \neg e))$$

would become

```
KB=[[a=-b]+~[c-=-[d*~e]]]
```

Note that we had to change the (bi-)implication and negation notation a little bit, since BASH does not accept `<`, `>` and `!`, since they all have a different usage within the shell. The before presented input generates a CNF expression

$$\begin{aligned}&(((((d \vee c) \vee (\neg a \vee b)) \wedge ((e \vee c) \vee (\neg a \vee b)))\\ &\wedge((\neg c \vee c) \wedge (\neg a \vee b))) \wedge ((((d \vee (\neg d \vee \neg e))\\ &\vee(\neg a \vee b)) \wedge ((e \wedge (\neg d \vee \neg e)) \vee (\neg a \vee b)))\\ &\wedge((\neg c \vee (\neg d \vee \neg e)) \vee (\neg a \vee b))))\end{aligned}$$

and yields the output

```
d,c,~a,b 10
e,c,~a,b 10
~c,c,~a,b 11
d,~d,~e,~a,b 14
e,~d,~e,~a,b 14
~c,~d,~e,~a,b 15
KB={[~a,b,c,d], [~a,b,c,e], [~a,b,c,~c]=TRUE,
[~a,b,d,~d,~e]=TRUE, [~a,b,~d,e,~e]=TRUE, [~a,b,~c,~d,~e]}
KB after resolution={[~a,b,c,d], [~a,b,c,e],
[~a,b,c,~c]=TRUE, [~a,b,d,~d,~e]=TRUE, [~a,b,~d,e,~e]=TRUE,
[~a,b,~c,~d,~e], [~a,b,d], [~a,b,c,~e], [~a,b,c,e,~e]=TRUE,
[~a,b,c,~c,~e]=TRUE, [~a,b,e], [~a,b,c,d,~d]=TRUE, [~a,b,c,~d],
[~a,b,c,~c,~d]=TRUE, [~a,b,~d,~e], [~a,b,e,~e]=TRUE,
[~a,b,d,~d]=TRUE, [~a,b,~c,~e], [~a,b,~c,~d], [~a,b,d,~e],
[~a,b,c], [~a,b,~d,e], [~a,b,~e], [~a,b,~d], [~a,b], [~a,b,~c]}
Resolution proof failed.
```

As expected, this sentence did not have any meaning. To get a resolvable sentence, one could put all premises in conjuction, as one statement (like this: `KB=[[..]*[..]*...]`). Our program does NOT support comma's to sepparate premises, so this a substitute for that expression.
The program itself is simple in essense. To recognize the expressions, we posed a grammar, which is written as a comment within the code for those interested. By implementing that, we generate an expression tree, which we can then transform into CNF. The premises can then be extracted from the CNF.