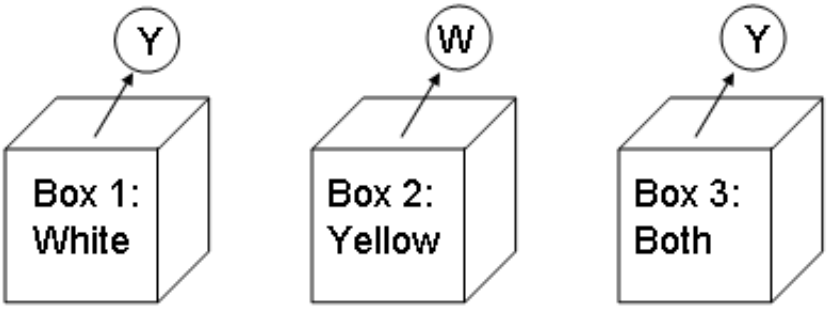
CSCE 420 - Spring 2021
Programing Assignment #3
due: Mon, Mar 22, 2021, 12:00pm (noon)

Overview

The goal of this project is to write a C++ program to do logical proofs using Natural Deduction. We will use **Sammy's Sport Shop** as an example, with a target inference to prove by Natural Deduction. You will have to write the propositional knowledge base (KB) for this problem, using a syntax to be defined. Code will be provided for reading in KBs and parsing sentences. You will then need to write functions for various propositional rules of inference (e.g Implication Elimination, Modus Ponens, Resolution, DeMorgan's Laws…). Finally, you will apply these inference steps to sentences from your KB to produce a derivation of the target inference for this problem.

Sammy's Sport Shop

You are the proprietor of *Sammy's Sport Shop*. You have just received a shipment of three boxes filled with tennis balls. One box contains only yellow tennis balls, one box contains only white tennis balls, and one contains both yellow and white tennis balls. You would like to stock the tennis balls in appropriate places on your shelves. Unfortunately, the boxes have been labeled incorrectly; the manufacturer tells you that you have exactly one box of each, but that **each box is definitely labeled wrong**. One ball is drawn from each box and its color observed. Given the initial (incorrect) labeling of the boxes above, and the three observations, use Propositional Logic to derive the correct labeling of the middle box.



Use propositional symbols in the following form: O1Y means a yellow ball was drawn (observed) from box 1, L1W means box 1 was initially labeled white, C1W means box 1 contains (only) white balls, and C1B means box 1 actually contains both types of tennis balls.

The initial facts describing this particular situation are: {O1Y, L1W, O2W, L2Y, O3Y, L3B}

1. Using these propositional symbols, write a propositional knowledge base (sammys.kb) that captures the knowledge in this domain (i.e. implications of what different observations or labels mean, as well as constraints inherent in this problem, such as that all boxes have different contents). *Do it in a complete and general way*, writing down *all* the rules and constraints, not just the ones needed to make the specific inference about the middle box. *Do not include derived knowledge* that depends on the particular labeling of this instance shown above.

2. Prove that box 2 must contain white balls (**C2W**) using Natural Deduction.

Format for KB Files

To represent propositional files, we will use a ASCII-based syntax called 'symbolic expressions'. In this syntax, there are only symbols and parentheses (for representing hierarchical/nesting structure).

- Symbols include any sequence of characters or digits (or some special chars, like '_'). For example, 'A12B', 'leftTurn', '3.1415', 'not', and 'implies' are symbols. These can be atomic sentences (propositions) by themselves; operator names are also symbols.
- Logical operators are represented as lists using prefix notation, that is, a sequence of symbols or sublists surrounded by parentheses, with the operator listed first. For example, **'(and P Q)'**, **'(not X)**', and '**(or (not X) (and P Q))**'. While the unary 'not' operator always has only 1 argument, 'and' and 'or' operators may have an arbitrary number of arguments (not restricted to binary).
- Implications are written similarly as lists using the 'implies' operator, such as "**(implies (and P Q) R)**", with 2 arguments: antecedent, followed by consequent.
- Extra white space doesn't matter; also, we will assume the parser is *case-insensitive*.
- KB files can also have blank lines, and lines beginning with '#' are assumed to be comments.

Examples:
```
(implies (and L M) P)
(or 2b (not 2b))
```

Expr Class – Parser for Logic Sentences

The internal representation for sentences is based on the **Expr class** (provided in Github via Proj3/parser.cpp and .hpp). There are actually 2 kinds of Expr objects: an ATOM (represented by the string 'sym'), and a LIST, represented by a vector of sub-expressions (vector<Expr*>) called 'sub'. This hierarchical data structure is intended to mimic the nested structure of the sentence syntax, but without the parentheses.

```
class Expr
{
public:
  EXPR_TYPE kind; // either ATOM or LIST
  string sym;
  vector<Expr*> sub;
  int end;

  Expr(string s);
  Expr(vector<string> tokens,int start);
  Expr(Expr* e); // deep copy
  Expr(vector<Expr*> L); // create new COMPLEX expression from list
  string toString();
};
```

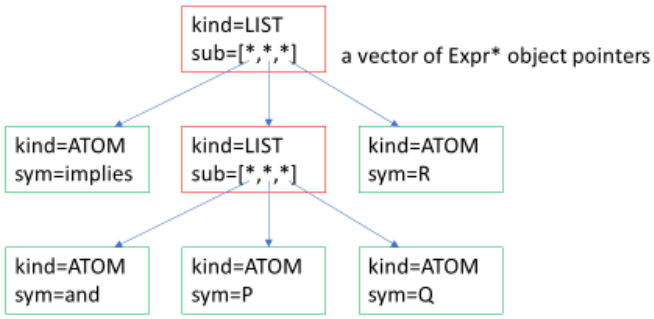
Instances of the Expr class are chimerical objects, depending on the value of 'kind':

- If kind==ATOM, then string *sym* is defined, and vector *sub* is undefined
- If kind==LIST, then vector *sub* is defined, and string *sym* is undefined

To make things easy, there is a **parse()** function provided for you, that takes a sentence as a string, constructs an Expr object for it, and returns a pointer to it. There is also a **toString()** method which serializes and Expr object back out to a string for printing. Here is an example snippet of C++ illustrating the use of the parser:

```
Expr* s1=parse("(implies P Q)");
Expr* s2=parse("P")
cout << s1->toString(); // output: (implies P Q)
```

Here is a visualization of the object structure created for "(implies (and P Q) R)":



If anything goes wrong during parsing, the parser code will throw an exception. There are two custom types of errors, derived from the C++ *runtime_error* class: SyntaxError, and RuleApplicationError (defined in parser.hpp). You can throw the latter exception if you are trying to apply an ROI to expression for which it doesn't apply (for example, trying to apply double-negation elimination to "(not P)"). In main, you should probably catch these exceptions and print out the message. Something like this…

```
int main()
{
  try {
    vector<Expr*> KB=load_KB();
    do_inference(KB);
  } catch (runtime_error& e) { cout << e.what() << endl; }
}
```

There is also an auxiliary function for reading in a whole KB file and returning a vector of Expr* objects:

```
vector<Expr*> load_kb(string fname);
```

Implementing Rules of Inference

Propositional Rules of Inference (ROI) can be implemented as functions that take 1 or 2 sentences (as Expr* objects), and return a new sentence (or possibly a list of sentences as vector<Expr*>). Using the examples above, here is how to doing Modus Ponens looks like:

```
Expr* s3=ModusPonens(s1,s2); // args: P->Q, and P
cout << s3->toString(); // output: Q
```

The way the ModusPonens() function would work is to check that the s2 expression matches the antecedents of the rule (by extracting the second element of the list for s1 and doing a hierarchical comparison via tree traversal with s2 to confirm they are identical, and then extracting the third complement of s1 and returning a copy of it.) This means you have to write some functions to check for equality of Expr objects and make copies.

Here are some other rules (ROI) you will have to implement:

- ModusPonens
- ImplicationElimination
- AndElimination
- AndIntroduction
- OrIntroduction
- DoubleNegationElimination – this is an easy one: “(not (not P))” -> P
- Resolution – this requires both input expressions to be disjunctions (‘or’s)
- DeMorgan’s Law(s) – 2 forms (‘not’ over ‘or’, ‘not’ over ‘and’)
- anything else you might want or need that is a sound ROI…

For each ROI you implement, include a test function, like the one shown above for Modus Ponens.

Doing a Natural Deduction Proof

A Natural Deduction proof can be done by reading in an initial set of sentences (premises) and then applying a sequence of ROI to generate new sentences, until you generate the one you are looking for. Determining which steps to perform is (admittedly) a manual process. (We will implement an automated proof search in a subsequent project). Consider the example shown in 7.16a in the textbook.

- P→Q
- L^M→P
- B^L→M
- A^P→L
- A^N→L
- A
- B

Q should be entailed by this KB. Here is one way to derive it using Nat Ded in C++:

example.kb:

```
# example from textbook, sentences 0 through 6:
(implies P Q)
(implies (and L M) P)
(implies (and B L) M)
(implies (and A P) L)
(implies (and A B) L)
A
B
```

inference_example.cpp

```
vector<Expr*> KB=load_KB("example.kb"); // parses and pushes sentences 0-6
KB.push_back(AndIntro(vector<Expr*>({KB[5],KB[6]})));  // 7. (and A B)
KB.push_back(ModusPonens(KB[4],KB[7]));                // 8. L
KB.push_back(AndIntro(vector<Expr*>({KB[6],KB[8]})));  // 9. (and B L)
KB.push_back(ModusPonens(KB[2],KB[9]));                // 10. M
KB.push_back(AndIntro(vector<Expr*>({KB[8],KB[10]}))); // 11. (and L M)
KB.push_back(ModusPonens(KB[1],KB[11]));               // 12. P
KB.push_back(ModusPonens(KB[0],KB[12]));               // 13. Q
```

If you print the final KB, here is what it looks like:

```
0. (implies P Q)
1. (implies (and L M) P)
2. (implies (and B L) M)
3. (implies (and A P) L)
4. (implies (and A B) L)
5. A
6. B
7. (and A B)
8. L
9. (and B L)
10. M
11. (and L M)
12. P
13. Q
```

What to Turn In

Create a subdirectory in your (private) Github project for the class (on the TAMU Github server) called 'Proj3'. In Proj3/, you should have at least the following files:

- **README** – should explain how your program works (e.g. command-line arguments), and any important parameters or constraints
- **makefile** – we must be able to compile your C++ code on compute.cs.tamu.edu by simply calling 'make'
- KB file: **sammys.kb**
- C++ files:
    - **NatDed.cpp**: contains your implementations of the Rules of Inference, to be linked into the main program below
    - **sammys.cpp** – this should each have a function that does a Natural Deduction proof for the target sentence by applying a sequence of Rules of Inference, and print out the final KB at the end
- transcripts: **sammys.txt** – show the final set of generated sentences

The date you commit your files and push them to the TAMU Github server will be used to determine when it is turned in and whether any late penalties will be applied.

Grading

The materials you turn in will be graded according to the following criteria:

- 25% - does it compile and run without problems on compute.cs.tamu.edu?
- 25% - does the implementation (code) look correct? (e.g. Rules of Inference)
- 25% - does the knowledge base file look correct? (all the right rules?)
- 25% - does the transcript look correct? (desired proof)