# Lab 3: Knowledge Base

EECS 348

In this lab assignment, you are going to extend a knowledge base (KB) and an inference engine. The knowledge base supports three main interfaces: `Assert`, `Retract`, and `Ask`.

- `Assert`: Add facts or rules into the knowledge base. After you add facts or rules into the KB, the forward-chaining algorithm is used to infer other facts or rules.
- `Ask`: ask queries and return a list of bindings for facts.
- `Retract`: remove facts from the knowledge base. Also, remove all other facts or rules that are dependent on the removed fact or rule.

## Starter code

We provide you five files with code: `main.py`, `logical_classes.py`, `read.py`, `util.py` and `student_code.py`. (Details about these files are described at the end of this write-up.)

- `main.py` contains code for testing the KnowledgeBase
- `student_code.py` contains the `KnowledgeBase` and `InferenceEngine` classes, and is where you will be writing code.
- `logical_classes.py` contains classes for each type of logical component, e.g. `Fact`, `Rule`, etc.
- `util.py` contains several useful helper functions
- `read.py` contains functions that read statements from files or terminal. (You won't need to read/explore this file.)

There are also two data files: `statements_kb.txt` and `statements_kb2.txt`. These files contain the facts and rules to be inserted into the KB. The provided tests use `statements_kb2.txt`, and you may use `statements_kb.txt` to generate your own tests.

## Your task

To get you started, the `Assert` and `Ask` interfaces have been written - exposed via the `KnowledgeBase.kb_assert` and `KnowledgeBase.kb_ask` methods.

Your task is two-part:

1. Implement the forward-chaining inferences that occurs upon asserting facts and rules into the KB - i.e., implement the `InferenceEnginer.fc_infer` method.
2. Implement the `Retract` interface to remove facts from the KB - i.e., implement the `KnowledgeBase.kb_retract` method.

### Rule currying in `fc_infer`

The key idea is that we don't just infer new facts - we can infer new rules.

When we add a new fact to the KB, we check to see if it triggers any rule(s). When we add a new rule, we check to see if it's triggered by existing facts.

However, a rule might have multiple statements on its left-hand side (LHS), and we don't want to iterate each of these statements every time we add a new fact to the KB. Instead, we'll employ a cool trick. Whenever we add a new rule, we'll only check the first element of the LHS of that rule against the facts in our KB. (If we add a new fact, we'll reverse this - we'll examine each rule in our KB, and check the first element of its LHS against this new fact.) If there's a match with this first element, we'll add a new rule paired with *bindings* for that match.

For example, imagine a box-world. Consider a rule stating that if a box `?x` is larger than another box `?y`, and box `?x` is on box `?y`, then box `?y` is covered. Formally, that looks like:

```
((sizeIsLess(?y, ?x), on(?x, ?y)) => covered(?y))
```

Now imagine that we know that box `A` is bigger than box `B`; i.e. that we have the fact `sizeIsLess(B, A)` in the KB. The above rule then matches, with the bindings `((?x: A, ?y: B))`. With that binding in place, we can now infer a new rule that uses it:

```
(on(A, B)) => covered(B)
```

If we find the fact `on(A, B)` in the KB, then we could use this rule to infer the fact `covered(B)`. If we don't have that fact, however, we now have a simple rule that will let us make the inference easily if we see that fact in the future.

## Removing rules and facts inferred from a removed fact

When you remove a fact, you also need to remove all facts and rules that were inferred using this fact. However, a given fact/rule might be supported by multiple facts - so, you'll need to check whether the facts/rules inferred from this fact are also supported by other facts (or if they were directly asserted).

As a simplification, you can assume that **no rules will create circular dependencies**. E.g., imagine a situation like `A => B`, `B => C`, and `C => B`. Removing `A` would mean removing `B` and `C`, since they depend on `A` via those rules. However, implementing that would get messy, since `B` and `C` depend on each other. You will **NOT** be given scenarios like this.

## Testing

To test this lab, we'll create several testing files that contain a bunch of facts and rules (similar to the ones provided). Each fact/rule will be asserted one-by-one into the KB. Other files containing more facts/rules will be used to test the `Retract` operation, and make sure it worked correctly. *We recommend that you manually check each function and make sure you understand each function and output.* **We also recommend you make your own testing files, and feel free to share them on Piazza.**. When sharing tests, please provide your rationale to the test, explain what you hope to test and/or how you developed the test.

## Hints

### Implementing `fc_infer`

- Use the `util.match` function to do unification and create possible bindings
- Use the `util.instantiate` function to bind a variable in the rest of a rule
- `Rule`s and `Fact`s have fields for `supported_by`, `supports_facts`, and `supports_rules`. Use them to track inferences! For example, imagine that a fact `F` and rule `R` matched to infer a new fact/rule `fr`.
  - `fr` is *supported* by `F` and `R`. Add them to `fr`'s `supported_by` list - you can do this by passing them as a constructor argument when creating `fr`.

  - `F` and `R` now *support* `fr`. Add `fr` to the `supports_rules` and `supports_facts` lists (as appropriate) in `F` and `R`.

**Implementing `kb_retract`**

- You can only remove an unsupported fact.
- Use the `supports_rules` and `supports_facts` fields to find and adjust facts and rules that are supported by a retracted fact.
  - The `supported_by` lists in each fact/rule that it supports needs to be adjusted accordingly.
  - If a supported fact/rule is no longer supported as a result of retracting this fact (and is not asserted), it should also be removed.

# Appendix: File Breakdown

Below is a description of each included file and the classes contained within each including a listing of their attributes. Each file has documentation in the code reflecting the information below (in most cases they are exactly the same). As you read through the attributes follow along in the corresponding files and make sure you're understanding the descriptions.

Attributes of each class are listed in the following format (*Note:* if you see a type like `Fact|Rule` the `|` type is `or` and mean the type can be either Fact or Rule):

- `field_name` (`type`) - text description

## logical_classes.py

This file defines all basic structure classes.

### Fact

Represents a fact in our knowledge base. Has a statement containing the content of the fact, e.g. (isa Sorceress Wizard) and fields tracking which facts/rules in the KB it supports and is supported by.

**Attributes**

- `name` (`str`): 'fact', the name of this class
- `statement` (`Statement`): statement of this fact, basically what the fact actually says
- `asserted` (`bool`): flag indicating if fact was asserted instead of inferred from other rules in the KB
- `supported_by` (`listof Fact|Rule`): Facts/Rules that allow inference of the statement
- `supports_facts` (`listof Fact`): Facts that this fact supports
- `supports_rules` (`listof Rule`): Rules that this fact supports

### Rule

Represents a rule in our knowledge base. Has a list of statements (the LHS) containing the statements that need to be in our KB for us to infer the RHS statement. Also has fields tracking which facts/rules in the KB it supports and is supported by.

**Attributes**

- `name` (`str`): 'rule', the name of this class
- `lhs` (`listof Statement`): LHS statements of this rule
- `rhs` (`Statement`): RHS statment of this rule
- `asserted` (`bool`): flag indicating if rule was asserted instead of inferred from other rules/facts in the KB
- `supported_by` (`listof Fact|Rule`): Facts/Rules that allow inference of the statement
- `supports_facts` (`listof Fact`): Facts that this rule supports
- `supports_rules` (`listof Rule`): Rules that this rule supports

### Statement

Represents a statement in our knowledge base, e.g. (attacked Ai Nosliw), (diamonds Loot), (isa Sorceress Wizard), etc. These statements show up in Facts or on the LHS and RHS of Rules.

**Attributes**

- `predicate` (`str`) - the predicate of the statement, e.g. isa, hero, needs

- `terms` (`listof Term`) - list of terms (Variable or Constant) in the statement, e.g. `'Nosliw'` or `'?d'`

#### Term

Represents a term (a Variable or Constant) in our knowledge base. Can sorta be thought of as a super class of Variable and Constant, though there is no actual inheritance implemented in the code.

**Attributes**

- `term` (`Variable|Constant`) - The Variable or Constant that this term holds (represents)

#### Variable

Represents a variable used in statements, e.g. `?x`.

**Attributes**

- `element` (`str`): The name of the variable, e.g. `'?x'`

#### Constant

Represents a constant used in statements

**Attributes**

- `element` (`str`): The value of the constant, e.g. `'Nosliw'`

#### Binding

Represents a binding of a constant to a variable, e.g. `'Nosliw'` might be bound to `'?d'`

**Attributes**

- `variable` (`str`): The name of the variable associated with this binding, e.g. `'?d'`
- `constant` (`str`): The value of the variable, e.g. `'Nosliw'`

#### Bindings

Represents Binding(s) used while matching two statements

**Attributes**

- `bindings` (`listof Bindings`) - bindings involved in match
- `bindings_dict` (`dictof Bindings`) - bindings involved in match where key is bound variable and value is bound value, e.g. some_bindings.bindings_dict[‘?d’] => ‘Nosliw’

**Methods**

- `add_binding(variable, value)` (`(Variable, Constant) => void`) - add a binding from a variable to a value
- `bound_to(variable)` (`(Variable) => Variable|Constant|False`) - check if variable is bound. If so return value bound to it, else False
- `test_and_bind(variable_verm,value_term)` (`(Term, Term) => bool`) - Check if variable_term already bound. If so return whether or not passed in value_term matches bound value. If not, add binding between variable_terma and value_term and return True.

#### ListOfBindings

Container for multiple Bindings

**Methods**

- `add_bindings(bindings, facts_rules)` - (`((Bindings, listof Fact|Rule) => void`) - add given bindings to list of Bindings along with associated rules or facts

## read.py

This file has no classes but defines useful helper functions for reading input from the user or a file.

**Functions**

- `read_tokenize(file)` - (`((str) => (listof Fact, listof Rule)`) - takes a filename, reads the file and returns a fact list and rule list.
- `read_from_input(message)` - (`((str) => str`) - collects user input from the command line.
- `parse_input(e)` - (`((str) => (int, str | listof str)`) - parses input, cleaning it as it does and assigning labels
- `get_new_fact_or_rule()` - (`(() => Fact | Rule`) - get a new fact or rule by typing, nothing passed in, data comes from user input
- `get_new_statements()` - (`(() => listof Statement`) - read statements from input, nothing passed in, data comes from user input

## util.py

This file has no classes but defines useful helper functions.

**Functions**

- `is_var(var)` (`((str|Variable|Constant|Term) => bool`) - check whether an element is a variable (either instance of Variable or string starting with `'?'`, e.g. `'?d'`)
- `match(state1, state2, bindings=None)` (`(Statement, Statement, Bindings) => Bindings|False`) - match two statements and return the associated bindings or False if there is no binding
- `match_recursive(terms1, terms2, bindings)` (`(listof Term, listof Term, Bindings) => Bindings|False`) - recursive helper for match
- `instantiate(statement, bindings)` (`(Statement, Bindings) => Statement|Term`) - generate Statement from given statement and bindings. Constructed statement has bound values for variables if they exist in bindings.
- `vprint(message, level, verbose, data=[])` (`(str, int, int, listof any) => void`) - prints message if verbose > level, if data provided then formats message with given data

## student_code.py

This file defines the two classes you must implement, KnowledgeBase and InferenceEngine.

#### KnowledgeBase

Represents a knowledge base and implements the three actions described in the writeup (`Assert`, `Retract` and `Ask`)

**InferenceEngine**

Represents an inference engine. Implements forward-chaining in this lab.