PROYECTO #2

(Ingeniería de Sistemas e Ingeniería en Telecomunicaciones)
Sergio Parra

Facultad de Ingeniería, Pontificia Universidad Javeriana
SISTEMAS 4800-4 (7238): introduccion a la IA (Teo-Prac)
José Alejandro León Andrade
22 de Agosto de 2025

1. Arquitectura general:
   El proyecto está organizado en módulos independientes con responsabilidades claras:

❖ main.py: entrada del programa, lectura de archivos de casos (cláusulas), orquestación de ejecución y guardado de resultados.

❖ resolution.py: algoritmo de resolución por refutación, funciones auxiliares para parseo de literales, aplicación de sustituciones y generación de resolventes. Depende de unify.unify.

❖ unify.py: implementación del algoritmo de unificación (incluye occurs-check y manejo de variables).

❖ fnc.py: utilidades para convertir sentencias de lógica de primer orden a Forma Normal Conjuntiva, contiene una función convertir_a_fnc que devuelve los pasos realizados. Puede ser ampliado para cubrir todas las reglas pedidas por la especificación.

2. Formato de entrada / salida

➢ Entrada: archivos de texto donde cada línea es una cláusula en FNC y literales separados por ∨ o v. main.py normaliza, elimina numeraciones iniciales y devuelve una lista de cláusulas representadas como set de literales.

➢ Salida: se generan archivos en la carpeta resultados/ con nombre <Caso>_resultado.txt que contienen la base de conocimiento leída y el "paso a paso" de las resoluciones realizadas durante la búsqueda por refutación. Ejemplos de resultados generados: Ancestor_Anna_John_resultado.txt y marco_cesar_resultado.txt. Estos muestran la traza y el resultado final (teorema demostrado o no).

3. Descripción de funciones

3.1 main.py: Orquestador ()

- cargar_clausulas(ruta): abre el archivo ruta, recorre línea por línea:

    - Si el archivo contiene cláusulas en FNC (líneas con literales separados por ∨ o v), las procesa como antes.

    - Si detecta una fórmula con cuantificadores (∀, ∃) o con conectivos (->, ∧, ∨), invoca convertir_a_fnc de fnc.py y obtiene las cláusulas resultantes automáticamente.

    - Muestra la traza de conversión en consola para documentar el proceso paso a paso (eliminación de implicaciones, Skolemización, etc.).

- ejecutar_caso(nombre, ruta): llamada principal por caso que:

    - carga KB con cargar_clausulas.

    - llama a resolution(kb) y recibe (resultado, pasos).

    - imprime pasos en consola y los guarda en txt con una estructura clara (KB, pasos, resultado final).

    - crea el directorio resultados si no existe.

3.2 resolution.py — Motor de resolución ()

- negar_literal(literal): devuelve la cadena literal negada. Si comienza con ¬ la quita, si no la antepone. Utilizado cuando se necesita construir el literal complementario.

- parse_literal(literal): parsea un literal string tipo '¬P(x,y)' o 'P(x,y)' en una tupla (predicado, [arg1, arg2, ...], negado_bool). Se usa para comparar predicados y extraer argumentos para unificación.

- aplicar_sustitucion(literal, sustituciones): dada una sustitución (dict mapping variable→valor), aplica la sustitución sobre los argumentos del literal y reconstruye el literal (con el prefijo ¬ si corresponde).

- resolve(clause1, clause2): intenta aplicar la regla de resolución entre todas las parejas de literales lit1 ∈ clause1, lit2 ∈ clause2:
    - Si ambos literales pertenecen al mismo predicado (mismo nombre) y tienen signos opuestos (uno negado, otro no) y misma aridad, llama a unify con las tuplas de argumentos.
    - Si unify devuelve una sustitución válida, construye la nueva cláusula resolvente: (clause1 ∪ clause2) \ {lit1, lit2} y aplica la sustitución a los literales restantes.
    - Retorna lista de resolventes (cada una como set de literales aplicados).
    - aplicar_sustitucion utiliza el diccionario retornado por unify asumiendo que keys son variables simples.
- resolution(kb): algoritmo principal (BFS-ish por pares):
    - Convierte la KB a lista de set.
    - Crea la lista pares con todas las combinaciones i<j.
    - Por cada par, obtiene resolventes y los añade a pasos (traza textual). Si aparece la cláusula vacía set() retorna True y la traza (teorema demostrado).
    - Mantiene new como conjunto de resolventes generados (como frozenset) y, si no genera cláusulas nuevas respecto a KB, termina con False.
    - Añade las nuevas cláusulas a la KB y repite.
    - es un algoritmo simple de saturación sin heurísticas (puede saturarse en cantidad de cláusulas).

3.3 unify.py: Algoritmo de unificación ()

- unify(x, y, substitutions=None): función recursiva que intenta unificar x e y. Comportamiento:

    - Si x == y retorna el substitutions acumulado.
    - Si x es string y x.islower() (se asume variable si está en minúscula), delega a unify_var.
    - Mismo para y.
    - Si ambos son tuplas de la misma longitud, unifica componente a componente.
    - None si no unificable.

- unify_var(var, x, substitutions): maneja mapeo variable valor:
    - Si var ya tiene sustitución, unifica la sustitución existente con x.
    - Evita ciclos con occurs_check.
    - Añade var: x al mapa y retorna el mapa.
- occurs_check(var, x, substitutions): evita unificación infinita comprobando si var aparece dentro de x (considerando tuplas).
- Observaciones sobre variables: la implementación identifica variables por str.islower() — por lo tanto las variables deben estar en minúsculas (ej.: x, y). Si en los casos las variables usan otro formato, la unificación puede fallar o comportarse inesperadamente.

3.4 fnc.py : Conversión a FNC ()

- eliminar_implicaciones(expr): transforma cada A -> B en (¬A ∨ B).
- mover_negaciones(expr): aplica leyes de De Morgan, elimina dobles negaciones y mueve las negaciones hacia los literales atómicos.
- estandarizar_variables(expr): renombra variables para evitar colisiones entre cláusulas.

- skolemizar(expr): elimina cuantificadores existenciales, sustituyendo por constantes o funciones de Skolem según corresponda.

- eliminar_cuantificadores(expr): remueve los cuantificadores universales (ya no son necesarios en FNC).

- distribuir_or_sobre_and(expr): distribuye las disyunciones sobre conjunciones para obtener la forma conjuntiva.

- extraer_clausulas(expr): transforma la fórmula final en una lista de cláusulas (conjuntos de literales) listas para el motor de resolución.

- convertir_a_fnc(expr): función principal que ejecuta todas las etapas anteriores y devuelve:

    - la lista de cláusulas en FNC

    - una traza textual con todos los pasos de la conversión

4. Pruebas

marco-cesar:

```
# Base de conocimiento: Marco odia a César

1. Hombre(Marco)
2. Pompeyano(Marco)
3. ¬Pompeyano(x3) ∨ Romano(x3)
4. Gobernante(Cesar)
5. ¬Romano(x5) ∨ Leal(x5, Cesar) ∨ Odia(x5, Cesar)
6. ¬Hombre(x6) ∨ ¬Gobernante(y6) ∨ ¬IntentaAsesinar(x6, y6) ∨ ¬Leal(x6, y6)
7. IntentaAsesinar(Marco, Cesar)

# Hipótesis: Odia(Marco, Cesar)
# Negación de la hipótesis:
8. ¬Odia(Marco, Cesar)
```

```
C:\Users\soros\OneDrive\Desktop\Proyecto2IA>main.py
Resultado guardado en: resultados\marco_cesar_resultado.txt
Resultado guardado en: resultados\teorema_ancestor_resultado.txt
```


```
Se derivó la cláusula vacía. Teorema demostrado.

Resultado final:
El teorema fue demostrado
```

Ancestor:

```
# Teorema lógico: "Anna es ancestro de John"
# Enunciado: Si Anna es madre de Bob, Bob de Carol y Carol de John,
# entonces Anna es ancestro de John.
# Fuente: ejemplo clásico de inferencia lógica (Herbrand, 1930)

1. Parent(Anna,Bob)
2. Parent(Bob,Carol)
3. Parent(Carol,John)

# Regla 1: Todo padre es ancestro
4. ¬Parent(x,y) ∨ Ancestor(x,y)

# Regla 2: Si x es padre de y y y es ancestro de z, entonces x es ancestro de z
5. ¬Parent(x,y) ∨ ¬Ancestor(y,z) ∨ Ancestor(x,z)

# Negación de la hipótesis (queremos refutarla)
6. ¬Ancestor(Anna,John)
```

```
Base de conocimiento:
{'Parent(Anna,Bob)'}
{'Parent(Bob,Carol)'}
{'Parent(Carol,John)'}
{'Ancestor(x,y)', '¬Parent(x,y)'}
{'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'}
{'¬Ancestor(Anna,John)'}

Pasos de resolución:
Resolviendo {'Parent(Anna,Bob)'} y {'Ancestor(x,y)', '¬Parent(x,y)'} -> {'Ancestor(Anna,Bob)'}
Resolviendo {'Parent(Anna,Bob)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'Ancestor(Anna,z)', '¬Ancestor(Bob,z)'}
Resolviendo {'Parent(Bob,Carol)'} y {'Ancestor(x,y)', '¬Parent(x,y)'} -> {'Ancestor(Bob,Carol)'}
Resolviendo {'Parent(Bob,Carol)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'Ancestor(Bob,z)', '¬Ancestor(Carol,z)'}
Resolviendo {'Parent(Carol,John)'} y {'Ancestor(x,y)', '¬Parent(x,y)'} -> {'Ancestor(Carol,John)'}
Resolviendo {'Parent(Carol,John)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'¬Ancestor(John,z)', 'Ancestor(Carol,z)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'Ancestor(y,z)', '¬Parent(y,z)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'¬Ancestor(Anna,John)'} -> {'¬Parent(Anna,John)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'¬Ancestor(Anna,John)'} -> {'¬Parent(Anna,y)', '¬Ancestor(y,John)'}
Resolviendo {'Parent(Anna,Bob)'} y {'Ancestor(x,y)', '¬Parent(x,y)'} -> {'Ancestor(Anna,Bob)'}
Resolviendo {'Parent(Anna,Bob)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'Ancestor(Anna,z)', '¬Ancestor(Bob,z)'}
Resolviendo {'Parent(Anna,Bob)'} y {'Ancestor(y,z)', '¬Parent(y,z)'} -> {'Ancestor(Anna,Bob)'}
Resolviendo {'Parent(Anna,Bob)'} y {'¬Parent(Anna,y)', '¬Ancestor(y,John)'} -> {'¬Ancestor(Bob,John)'}
Resolviendo {'Parent(Bob,Carol)'} y {'Ancestor(x,y)', '¬Parent(x,y)'} -> {'Ancestor(Bob,Carol)'}
Resolviendo {'Parent(Bob,Carol)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'Ancestor(Bob,z)', '¬Ancestor(Carol,z)'}
Resolviendo {'Parent(Bob,Carol)'} y {'Ancestor(y,z)', '¬Parent(y,z)'} -> {'Ancestor(Bob,Carol)'}
Resolviendo {'Parent(Carol,John)'} y {'Ancestor(x,y)', '¬Parent(x,y)'} -> {'Ancestor(Carol,John)'}
Resolviendo {'Parent(Carol,John)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'¬Ancestor(John,z)', 'Ancestor(Carol,z)'}
Resolviendo {'Parent(Carol,John)'} y {'Ancestor(y,z)', '¬Parent(y,z)'} -> {'Ancestor(Carol,John)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} -> {'Ancestor(y,z)', '¬Parent(y,z)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'¬Ancestor(Anna,John)'} -> {'¬Parent(Anna,John)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'¬Ancestor(John,z)', 'Ancestor(Carol,z)'} -> {'¬Parent(John,z)', 'Ancestor(Carol,z)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'Ancestor(Bob,z)', '¬Ancestor(Carol,z)'} -> {'Ancestor(Bob,z)', '¬Parent(Carol,z)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'¬Parent(Anna,y)', '¬Ancestor(y,John)'} -> {'¬Parent(Anna,John)', '¬Parent(y,John)'}
Resolviendo {'Ancestor(x,y)', '¬Parent(x,y)'} y {'Ancestor(Anna,z)', '¬Ancestor(Bob,z)'} -> {'Ancestor(Anna,z)', '¬Parent(Bob,z)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'¬Ancestor(Anna,John)'} -> {'¬Parent(Anna,y)', '¬Ancestor(y,John)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(Anna,Bob)'} -> {'Ancestor(x,Bob)', '¬Parent(x,Anna)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(Bob,Carol)'} -> {'¬Parent(x,Bob)', 'Ancestor(x,Carol)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(y,z)', '¬Parent(y,z)'} -> {'Ancestor(x,z)', '¬Parent(y,z)', '¬Parent(x,y)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(Carol,John)'} -> {'¬Parent(x,Carol)', 'Ancestor(x,John)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'¬Ancestor(John,z)', 'Ancestor(Carol,z)'} -> {'¬Parent(John,y)', 'Ancestor(Carol,z)', '¬Ancestor(y,z)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'¬Ancestor(John,z)', 'Ancestor(Carol,z)'} -> {'¬Parent(x,Carol)', '¬Ancestor(John,z)', 'Ancestor(x,z)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(Bob,z)', '¬Ancestor(Carol,z)'} -> {'¬Parent(Carol,y)', 'Ancestor(Bob,z)', '¬Ancestor(y,z)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(Bob,z)', '¬Ancestor(Carol,z)'} -> {'Ancestor(x,z)', '¬Parent(x,Bob)', '¬Ancestor(Carol,z)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'¬Parent(Anna,y)', '¬Ancestor(y,John)'} -> {'¬Ancestor(y,John)', '¬Parent(Anna,y)', '¬Parent(y,y)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(Anna,z)', '¬Ancestor(Bob,z)'} -> {'¬Parent(Bob,y)', 'Ancestor(Anna,z)', '¬Ancestor(y,z)'}
Resolviendo {'Ancestor(x,z)', '¬Ancestor(y,z)', '¬Parent(x,y)'} y {'Ancestor(Anna,z)', '¬Ancestor(Bob,z)'} -> {'¬Ancestor(Bob,z)', '¬Parent(x,Anna)', 'Ancestor(x,z)'}
'¬Ancestor(Anna,John)'} y {'Ancestor(y,z)', '¬Parent(y,z)'} -> {'¬Parent(Anna,John)'}
'¬Ancestor(Anna,John)'} y {'Ancestor(Anna,z)', '¬Ancestor(Bob,z)'} -> {'¬Ancestor(Bob,John)'}
```

```
Se derivó la cláusula vacía. Teorema demostrado.

Resultado final:
El teorema fue demostrado
```

5. Resultados

Durante la validación del sistema se ejecutaron dos conjuntos de pruebas principales:

- Caso Marco-César: Se introdujeron las cláusulas relacionadas con la lealtad, la ciudadanía romana y el intento de asesinato de César por parte de Marco. El motor de resolución aplicó sucesivamente unificación y resolución, generando una traza detallada hasta derivar la cláusula vacía.
    - Resultado: *El teorema fue demostrado*.
    - Evidencia: archivo marco_cesar_resultado.txt.

- Caso Ancestor (Anna–John): Se utilizaron cláusulas que modelan la relación de ancestros y descendientes. El proceso mostró la conversión a FNC, la expansión de la base de conocimiento y las resoluciones intermedias.
    - Resultado: *El teorema fue demostrado*.
    - Evidencia: archivo Ancestor_Anna_John_resultado.txt.

En ambos casos los archivos de salida muestran:

1. La base de conocimiento cargada (en FNC).
2. El paso a paso de cada resolución realizada.
3. El resultado final indicando si el teorema fue demostrado o no.

Los screenshots de estos .txt constituyen la prueba empírica del funcionamiento correcto del sistema.

6. Conclusiones

- ❖ El sistema implementado cumple con el objetivo planteado: verificar teoremas en lógica de primer orden mediante resolución por refutación.
- ❖ La organización modular permitió una separación clara de responsabilidades: unificación, resolución, conversión a FNC y control del flujo principal.

- La traza generada en los archivos de salida permite seguir con transparencia cada paso de la inferencia, lo que facilita tanto la validación del algoritmo como la explicación didáctica.
- La integración con fnc.py mejora la robustez, ya que el sistema puede trabajar tanto con cláusulas ya normalizadas como con fórmulas más generales que requieren conversión a FNC.
- Las pruebas realizadas muestran que el algoritmo es correcto: logra derivar la cláusula vacía cuando el teorema es verdadero, confirmando así la validez de la demostración.
- Una posible mejora futura es la optimización de la estrategia de selección de pares de cláusulas para evitar la explosión combinatoria en bases de conocimiento más grandes.

En conclusión, seimplementa de manera efectiva un motor de inferencia por resolución y genera resultados trazables.