.UBAfiuba
FACULTAD DE INGENIERÍA

Paradigmas de programación
(TB025) Curso Brasburg

Trabajo Práctico 1
Prolog en Scala

7 de octubre de 2025

Abel Tomas Modesti Funes	Jose Ignacio Adelardi	Guido Cuppari	Julieta Sosa Orsi
111932	111701	111172	109332

1. Compilación y ejecución de Tests:

Previo a la ejecución del programa debe verificarse tener instaladas las herramientas pertinentes:

- sbt (Scala Build Tool)
- java 8 o superior
- Sistema operativo Unix/Linux/macOS

1.1. Ejecución de múltiples tests:

La ejecución de todos los tests puede realizarse fácilmente con `./test_runner.sh`. Este script procede a compilar el código para luego ejecutar secuencialmente uno a uno los diferentes programas de ejemplos, tomando el programa con terminación `.pl` y procesandolo, luego toma las consultas del archivo `.txt` de mismo nombre para realizar las consultas indicadas, tras esto va a buscar el resultado esperado en el archivo de mismo nombre con terminación `.res` y compara lo devuelto por el programa con el resultado esperado e indica si es correcto o no (en este caso indicando que fue lo recibido y lo esperado).

1.2. Ejecución de un programa:

Para este caso desde la raiz del proyecto puede ejecutarse por la terminal mediante *sbt* con el siguiente comando:

```
sbt "run <path/de/entrada.pl> <path/de/input.txt>"
```

Opcionalmente se puede ejecutar con el input " − " para que el programa tome las consultas por $STDIN$. Este proceso termina cuando el usuario ingresa la consulta $EXIT$.

2. Análisis de Código

2.1. Funcionalidades Implementadas y Análisis del Código

El trabajo implementa un motor lógico similar a Prolog, que permite definir una base de conocimientos con hechos y reglas, y luego realizar consultas para verificar si algo es verdadero o falso según esa base. También soporta variables libres, por lo que puede encontrar valores que hagan cierta una proposición.

El programa se ejecuta por consola y recibe dos archivos como argumentos: uno con la base de conocimientos (en formato `.pl`) y otro con las consultas a evaluar. Cada línea del archivo de consultas produce una línea de salida, que puede ser simplemente `true` o `false`, o bien `true (X = valor)` cuando hay variables que se unifican.

El desarrollo se dividió en varios módulos para mantener una estructura clara:

- **Clases**: definen las estructuras principales del lenguaje lógico.
- **Parser**: se encarga de leer los archivos y transformarlos en objetos del programa.
- **Engine**: resuelve las consultas mediante unificación, sustituciones y backtracking.
- **Main**: gestiona la ejecución general, lee los archivos y muestra los resultados.

2.2. Partes del Código

2.2.1. Clases

En este módulo se definieron las entidades básicas del lenguaje:

- `Atomo`: representa una constante o símbolo, como por ejemplo `juan` o `auto`.
- `Variable`: representa variables libres o ligadas, que se escriben con mayúscula, como `X` o `Y`.
- `Predicado`: combina un nombre con una lista de expresiones, por ejemplo `padre(juan, maria)`.
- `Hecho`: representa un predicado simple, sin condiciones.
- `Regla`: representa una cláusula de Horn con cabeza y cuerpo, por ejemplo `abuelo(X,Y) :- padre(X,Z), padre(Z,Y)`.

Estas estructuras reflejan la forma en que Prolog interpreta la información, pero adaptadas a un modelo funcional en Scala. Todas son inmutables, lo que evita efectos secundarios y mantiene el código más claro.

2.2.2. Parser

El `Parser` es el encargado de leer las líneas de los archivos y convertirlas en objetos de tipo `Conocimiento`. Su funcionamiento se basa en reconocer la estructura de las sentencias y crear las instancias correspondientes. Las funciones más importantes son:

- **parseExpresion**: identifica si el token leído es un átomo, una variable o un predicado.
- **parsePredicado**: separa el nombre y los argumentos, teniendo en cuenta los paréntesis y las comas.
- **parseHecho** y **parseRule**: crean los objetos `Hecho` y `Regla` a partir de texto.

- **parseLines**: recibe una lista de líneas y devuelve una lista con todos los conocimientos válidos.

Además, se implementó una función auxiliar para dividir correctamente los argumentos separados por comas sin confundirse con los que están dentro de paréntesis. Esto es clave para que el parseo funcione bien incluso con reglas más complejas.

2.2.3. Engine / Unificación

El `Engine` es el corazón del programa. Aquí se resuelven las consultas aplicando el algoritmo de unificación, las sustituciones necesarias y el mecanismo de backtracking.

Algunas de las funciones más importantes son:

- **aplicarSustitucion**: reemplaza las variables por los valores que fueron ligados en una sustitución.
- **extraerVariables**: obtiene las variables libres que aparecen en una consulta.
- **standardizeApart**: renombra variables internas de una regla para evitar conflictos cuando se aplican varias veces.
- **evaluarBuiltIn**: maneja operaciones especiales como `suma/3`, `resta/3`, `multiplicacion/3`, `eq/2`, `neq/2` y `gt/2`. Estas operaciones se resuelven directamente en el motor, sin buscar en la base de conocimiento.
- **resolverMeta** y **resolverCuerpo**: implementan la lógica de resolución. Buscan hechos o reglas que unifiquen con la meta actual, aplican las sustituciones y continúan con las metas siguientes. Si falla una unificación, se retrocede (backtracking) y se prueba otra opción.
- **formatResultado**: prepara la salida final, mostrando las variables y los valores que las satisfacen.

La unificación se implementó de forma recursiva. Si ambas expresiones son iguales, no hace nada; si una es variable, la liga con la otra; si ambas son predicados con igual nombre y cantidad de argumentos, intenta unificarlos uno por uno. Si no coincide la estructura, la unificación falla.

3. Hipótesis

Durante el desarrollo del TP se establecieron ciertas hipótesis con el objetivo de delimitar el alcance del sistema y facilitar su implementación.

Se asumió que:

- Las bases de conocimiento están bien formadas y que las cláusulas se expresan de manera sintacticamente correcta.
- Los hechos y reglas ingresados no contienen recursión infinita ni construcciones que impidan la finalización del proceso de inferencia.
- La unificación implementada es suficiente para resolver los objetivos propuestos, sin necesidad de extenderla a operadores aritméticos o estructuras de datos complejas.
- Las variables son locales a cada consulta y que no se requiere un manejo de entornos anidados o contextos múltiples.
- Mundo cerrado: toda consulta que no pueda ser unificada o resuelta se considera falsa.
- Por ultimo, se consideró que la interacción con el usuario, ya sea por consola o interfaz, no afecta el funcionamiento lógico del motor y se limita a la carga y visualización de resultados.

4. Conclusiones

El desarrollo de este TP nos permitió entender de manera profunda el funcionamiento interno de un motor de inferencia basado en lógica de primer orden, similar al utilizado por Prolog. A través de la implementación en **Scala 3**, se exploraron conceptos fundamentales, tales como la unificación, la resolución de metas y el backtracking.

Durante la construcción del módulo logico en **engine.scala**, se logró implementar un sistema de resolución funcional que integra los componentes principales: el parser encargado del análisis sintactico de la base de conocimiento, el módulo de unificación que maneja las sustituciones entre términos, y el motor de inferencia que aplica recursivamente las reglas de resolución para encontrar soluciones válidas a las consultas.

La integración con **main.scala** permitió tener una interfaz de ejecución completa, capaz de leer una base de conocimiento, procesar consultas y mostrar resultados mediante sustituciones válidas. De esta forma, el motor desarrollado cumple con los principios del razonamiento lógico automatizado y demuestra un comportamiento equivalente al de un intérprete Prolog básico.

En términos de aprendizaje, este trabajo ayudo a la comprensión del paradigma lógico y su relación con los mecanismos de búsqueda y deducción, así como la importancia del manejo de variables, la estandarización de nombres y en general las buenas practicas de programacion. Además, el uso de Scala aportó una visión funcional del problema, facilitando la implementación de estructuras inmutables.

En conclusión, el trabajo permitió no solo construir un sistema operativo y coherente de inferencia lógica, sino también afianzar habilidades de diseño, abstracción y razonamiento formal en el contexto de la programación funcional.