TP2: Funco-Logi

Detalles del trabajo

Objetivos

- Analizar el problema planteado correctamente.
- Diseñar una solución utilizando correctamente los conceptos de **programación funcional**.
- Seguir las buenas prácticas de programación.

Indicaciones del trabajo

Grupos de trabajo

El trabajo se desarrollará de forma grupal, en grupos de máximo 4 alumnos. El trabajo práctico debe realizarse utilizando Git y Github como herramienta de colaboración.
Todos los alumnos integrantes del grupo **deben** participar activamente del desarrollo del trabajo de forma equitativa. Se espera que la participación sea equitativa, queda a criterio de cada grupo la forma de trabajo y organización.

Tecnología

El trabajo debe realizarse en *Scala 3*, utilizando cualquier versión estable de esta versión. Para comenzar el trabajo se debe clonar este repositorio y programar la solución siguiendo la estructura de los archivos. Seguir el *ReadMe* del repositorio para poder correr las pruebas provistas por la cátedra. Los archivos que ya se encuentran en el repositorio son mínimos, se espera que cada grupo agregue las pruebas que considere necesario.

Tiempo de trabajo

Se contarán con aproximadamente 4 semanas completas para llevar a cabo el desarrollo del TP. Transcurrido el tiempo previsto para la realización del trabajo práctico debe realizarse la **entrega**.

Modalidad de entrega

Para realizar la entrega se debe:

- Github:
  - El repositorio tiene que ser privado y debe estar su corrector invitado al mismo.
  - Para realizar la entrega se debe crear un branch llamado `tp-2`, con el código correspondiente a la entrega. No se debe modificar más luego de la fecha de entrega.

    - Se debe contar con un `README.md` que explique cómo correr el programa “desde cero”, cómo instalar las dependencias y otras explicaciones pertinentes. El código debe poder correrse con un comando que compile el código y lo ejecute, **no es válido subir un ejecutable y que el comando simplemente lo corra**. Las instrucciones deben ser genéricas, no pueden depender de un IDE puntual.
- Mail: Mandar un mail a algoritmos3.fiuba@gmail.com indicando número de grupo y sus integrantes, aclarando cuál es la branch y el repositorio de la entrega. El mail debe llevar adjunto el informe en formato PDF.

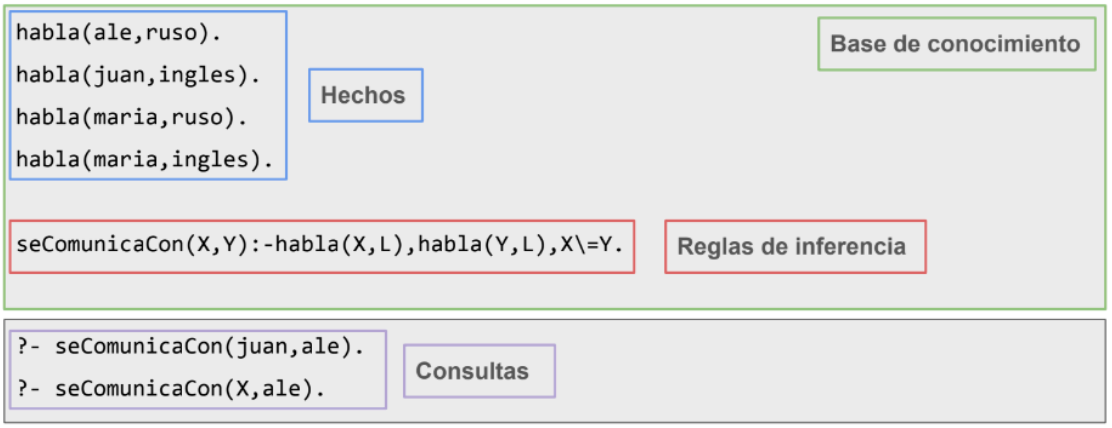
Fecha de entrega

La entrega definitiva, que debe alinearse a lo especificado en la Modalidad de entrega, debe realizarse con anterioridad a las 23:59hs del 19/11/2026.

Enunciado: Consultas de Prolog

Dominio

El trabajo consiste en implementar un motor lógico inspirado en Prolog. Los programas estarán compuestos por hechos y reglas, y el sistema deberá ser capaz de responder consultas mediante unificación, sustitución y resolución con DFS/backtracking.



El objetivo del trabajo es realizar una herramienta que contemple un subset de las mismas o similares funcionalidades de Prolog, teniendo un alcance acotado de la especificación. La herramienta debe ser realizada en Scala y basada en los conceptos de la programación funcional.
También, parte del objetivo de este trabajo es ofrecer una versión de Prolog que **únicamente** permita escribir reglas en formas de **cláusulas de Horn**.

Componentes

Nombre		Descripción	Ejemplo
Variables		Comienzan con mayúscula o _	X, Y
Números		Enteros	0, 2, 12
Átomos		Constantes, text, nombres de términos compuestos	algoritmo, “paradigmas de la programación”
Predicados (nombre + aridad)		n = 0 => Enunciados n = 1 => Propiedades n = 2 => Relaciones n > 2 => Relaciones n-arias	sol. vehículo(auto). cursa(ramiro, tda). profesor(fede, algo3, 2025).
Clausula	Hecho	pred(t1,...,tn).	divide(2,6).
	Reglas	Cabeza :- C1, C2, ..., Ck.	abuelo(X,Y) :- padre(X,Z), padre(Z,Y).
Consulta		Pregunta al programa	?- divide(2,6).

Requerimientos

Importante: En caso de tomar una decisión respecto a una definición ambigua del dominio o los requerimientos, volcarlo en la correspondiente sección de hipótesis del informe.

Entrada y argumentos

El programa al invocarlo debe recibir como argumentos el path (la ruta) al archivo de la base de conocimientos y al archivo que va a tener las consultas a realizar.

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

El archivo de input tiene por cada línea una consulta diferente por lo que se espera que el output del programa tenga la misma cantidad de líneas que el archivo de input. Se exige únicamente que el output se muestre por salida estándar.

Por comodidad o para repetir ejecuciones, sugerimos también ingresar la entrada via piping. Asimismo, se puede redirigir la salida del programa hacia un archivo, con el operador clásico de la shell:

```
$ funcologi programas/base.txt inputs/base.pl > <output file>
```

> *Nota: se puede compilar y generar un ejecutable para correr el programa, en lugar de llamar a sbt en cada corrida.*

Se deben interpretar y modelar internamente las reglas de la base de conocimientos, de manera que sepan cómo deben aplicarse según la consulta.

Sintaxis

Parseo

La sintaxis que soportamos en este trabajo es una versión simplificada de la ofrecida por SWISH-Prolog, donde se tomaron ciertas suposiciones para simplificar su parseo.

- Todas las sentencias terminan con un punto (`.`).
- Se puede suponer que no hay espacios dentro de las sentencias de la base de conocimientos (si se desea, pueden contemplarlos).
- Pueden haber infinitos saltos de línea entre las distintas sentencias.
- Las palabras iniciadas con mayúscula se reservan únicamente para representar variables.
- Las líneas de la base de conocimiento que comienzan con `%%` se consideran comentarios y deben ignorarse.
- Referido a los **hechos**:
    - Siempre cuentan con la siguiente forma:
        - Para aridad 0 → `nombre_hecho().`
        - Para aridad N > 0 → `nombre_hecho(arg1, …, argN).`

    - Pueden haber infinitas definiciones del mismo hecho, con diferentes parámetros.
- Referido a las **reglas de inferencia**:
    - Se considera que las reglas se definen en una única línea, no es necesario contemplar saltos de línea entre las distintas proposiciones lógicas.
    - Toda definición de regla sigue el siguiente formato:
        - `nombre_regla(arg1,…,argN):-prop1,prop2,…,propN.`
    - Si la regla no cuenta con argumentos, el paréntesis quedará vacío (también puede contemplarse la omisión, si se desea).
- Referido a **operaciones built-in** (ver encabezado más adelante):
    - Todas las operaciones built-in siguen el formato:
        - `nombre_operacion(param1,…,paramN)`

Operaciones built-in

Dado que complejizaría mucho el parseo del input del programa, se elimina cierta sintaxis de operaciones que ofrece SWISH-Prolog, tales como operaciones aritméticas, manipulación de listas, entre otras.

Sin embargo, para no perder funcionalidad, complementamos con operaciones built-in que poseen una sintaxis más sencilla, y nos permiten realizar un amplio espectro de operaciones para sumar y desarrollar nuestra base de conocimientos.

Por tanto, es necesario implementar las siguientes operaciones built-in del programa, que reservan un palabra clave (keyword) y al invocarse, se ejecuta un bloque de código concreto y es reemplazado por su resultado (estas son las únicas operaciones que **retornan** un resultado):

Operación	Ejemplo	Resultado
add/2	add(2, 3).	5
sub/2	sub(4, 3).	1
mult/2	mult(2, 5).	10
eq/2	eq(2, 3).	false
gt/2	gt(7,1).	true
eqStr/2	eqStr(“hola”, “chau”).	false
len/1	len(“hola”).	4
not/1	not(false) not(eq(len(“pepe”),4))	true false

NOTA: se pueden anidar operaciones built-in, como ocurre en el ejemplo de not

Variables

Las variables dentro de nuestro motor lógico pueden tener 2 “estados”: libres o asignadas.
Las variables asignadas son aquellas que vienen con un valor ya determinado de un punto previo en la ejecución de las reglas lógicas.
Las variables libres son aquellas que aún no tienen un valor asignado, y que por tanto el motor lógico puede ligar según el criterio de unificación lo permita, o mediante asignación.

La asignación en este motor lógico se dará utilizando el operador `'='`, donde siendo la sentencia `X=Y`, ligará a una determinada variable X la referencia de Y (que puede ser un valor, un término u otra variable).

Además, dentro de una regla de inferencia, puede sernos muy útil definir variables auxiliares que sirvan para conectar 2 hechos aislados. Estas variables nuevas, al ser declaradas, lógicamente serán libres, y pueden llegar a ligarse utilizando el operador `'='` o como parte parte de la unificación.

Ejemplos:

- Variable auxiliar Z, libre:
  - `esAbuelo(X,Y):-esPadre(X,Z),esPadre(Z,Y).`
- Variable auxiliar N1, asignada mediante `=`:
  - `puedeLlegar(X,Y,N):-N1=sub(N,1),puedeLlegar(X,Y,N1).`

Por simplicidad, para este trabajo no se busca contemplar la posibilidad de variables “omitibles” (e.x. `puedeLlegar(X,Y,_):-bondi(X,Y).`), con lo cual puede obviarse que se encuentre una regla o hecho similar en la base de conocimiento.

Consultas

El Funco-Logi debe soportar tanto consultas:

1. Sin asignación de variables: la consulta solamente busca la veracidad de un proposición, no busca que se identifique valores de variables.
   a. Ejemplo 1: `?- hoyLlueve()`
   b. Ejemplo 2: `?- sonAmigos(carlos,juan)`
2. Con asignación de variables: se deja libres ciertas variables, y busca saber si para algún valor de esas variables la consulta resulta verdadera. En caso de que así sea, se espera retornar aquel set de valores que hicieron que se considere veraz.
   a. Ejemplo 1: `?- aprueba(matematica, ALUMNO)`
   b. Ejemplo 2: `?- sonAmigos(P1,P2)`

Si existen múltiples soluciones, se deben encontrar todas las combinaciones de variables que resuelven y retornarlas, en el formato:

```
[(P1=carlos;P2=juan),(P1=carlos;P2=mariano)]
```

Criterio de veracidad

Solamente se considera verdadero lo que se encuentra de forma directa o indirecta en la base de conocimiento. Cualquier otro hecho no deducible de la base de conocimientos se considera como falso.

Tal como se vió en la teórica, la veracidad de una cláusula de Horn está dada por la veracidad conjunta de todas las proposiciones que la componen (i.e. una cadena de ANDs).

Al igual que en SWISH-Prolog, la existencia y veracidad relativa de los hechos se determina por el orden en que aparecen en la base de conocimientos. Es decir, si hay 2 hechos que cumplen cierta consulta o unifican en determinada regla, entonces el programa debe inclinarse por el primer hecho que aparece en la base (leído de arriba hacia abajo).

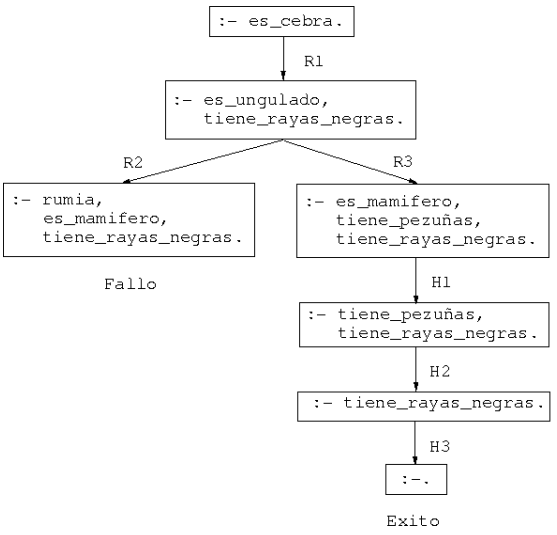
Reglas de inferencia

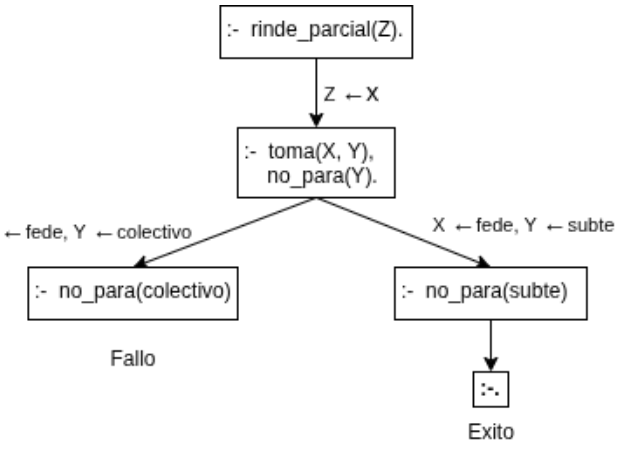
En líneas generales, la inferencia realizada por el motor lógica sigue la siguiente secuencia:

1. Dado un objetivo `?- G1, G2, ..., Gn.`, seleccionar `G1`.
2. Buscar en el programa, de arriba hacia abajo, una cláusula cuya cabeza unifique con `G1`.
3. Si unifica con sustitución σ, reemplazar `G1` por el cuerpo de la cláusula (si es hecho, cuerpo vacío) y aplicar σ al resto de metas.
4. Continuar recursivamente con el nuevo vector de metas, de izquierda a derecha.
5. Ante una falla, se aplica un backtracking al último punto de elección (otra cláusula que unifique) y continua la ejecución.
6. Cada vez que la lista de metas queda vacía, reportar una solución (las variables libres de la consulta con su sustitución).
7. Repetir hasta agotar alternativas.

Ejemplos

Base de conocimientos (nota: la sintaxis no es la simplificada, por temas de formato del documento)	**Ejecución**

`es_cebra():-`
`  es_ungulado(),`
`  tiene_rayas_negras().`
`es_ungulado():-`
`  rumia(),es_mamífero().`
`es_ungulado():-`
`  es_mamífero(),`
`  tiene_pezuñas().`
`rumia():- false.`
 	
`es_mamífero().`
`tiene_pezuñas().`
`tiene_rayas_negras().`
	
	Consulta: ?- es_cebra()
	true

`rinde_parcial(X):-`
`  toma(X,Y),no_para(Y).`
 	
`toma(fede,coletivo).`
`toma(fede,subte).`
`no_para(subte).`
	
	Consulta: ?- rinde_parcial(Z).
	true [(Z=fede)]

Informe

Se espera la realización de un informe que contenga:

- Cómo se compila y corre el programa.
- Qué funcionalidades fueron implementadas, con uno o más ejemplos que hayan utilizado para validar su correcta ejecución.
- Una sección de las hipótesis tomadas durante la realización del trabajo.
- Conclusiones.

Criterios de aprobación

Para que un trabajo práctico se considere aprobado, debe ser entregado en tiempo y forma, y debe funcionar correctamente para el siguiente listado de pruebas mínimas:

Pruebas mínimas

Las pruebas constan de 3 archivos `input.pl, consultas.txt` y `output.res`, donde:

- `input.pl` = es el archivo que contiene la base de conocimientos para nuestro programa.
- `consultas.txt` = posee N líneas, donde cada una es una consulta a realizar sobre la base de conocimientos.
- `output.res` = también posee N líneas, donde cada una es la respuesta a la consulta realizada en el archivo de consultas.
    - La respuesta es la veracidad de la query (true o false).
    - Adicionalmente, cuando hay variables libres en la consulta, se deben mostrar aquellos valores que toman las variables para que la query se considere verdadera.
    - Ejemplos:
        - `false`
        - `true [(Z=auto;C=rojo)]`