Laboratorio di Intelligenza Artificiale

Andrea Ciccarello

A.A. 2024-2025

Esercizio 1

Problema. Il problema consiste nel creare un ragionatore in logica proposizionale, sfruttando regole logiche. Queste regole sono espresse sotto forma di implicazioni logiche (ad esempio, *se A allora B*), e l'obiettivo del ragionatore è applicare tali regole in modo sistematico per estendere un insieme di condizioni iniziali fino a raggiungere le condizioni finali desiderate.

Il cuore di questo sistema è il motore inferenziale, un componente che applica meccanismi di inferenza per dedurre nuove informazioni a partire da un insieme di fatti e regole. Un ragionatore deve essere in grado di gestire le complessità delle regole logiche e applicare in modo efficiente le inferenze, rispettando la correttezza logica e garantendo la validità delle deduzioni. Il compito del programma in *Fabula* sarà quello di analizzare le regole (espresse come implicazioni) e applicarle per raggiungere le conclusioni desiderate, espandendo progressivamente il set di fatti noti.

Le implicazioni vengono espresse come una congiunzione di ipotesi (gli antecedenti) che generano dei fatti nuovi del tipo:

$$\text{antecedente1}_1 \land \text{antecedente1}_2 \land \ldots \land \text{antecedente1}_N \rightarrow \text{tesi1}_1$$

$$\text{antecedente2}_1 \land \text{antecedente2}_2 \land \ldots \land \text{antecedente2}_N \rightarrow \text{tesi2}_1$$

Esse vengono espresse con una lista di termini `impl(antecedente,tesi)`. Gli antecedenti vengono rappresentati tramite degli una lista di and. I goal sono rappresentati da uno o più proposizioni all'interno di un or logico. La signature della funzione ad alto livello, che successivamente verrà implementata nel linguaggio logico funzionale *Fabula* dovrà essere la seguente:

```
1 check([Lista di antecedenti e tesi],
2     [Lista di fatti veri],[Goal])
```

Il ragionatore ha 2 strade da poter intraprendere per poter risolvere questo problema:

1. *Ragionamento in avanti*: partendo da un certo insieme di regole generare tutte le nuove verità, partendo da un insieme di fatti veri.

2. *Ragionamento indietro*: con l'obiettivo di rendere vero un fatto, quali regole poter applicare e sotto quali condizioni posso portare a quel fatto.

Metodo. L'algoritmo alla base di check_fwd/3 è una variante della catena in avanti (forward chaining), uno dei due principali metodi di ragionamento usati da un motore inferenziale. Essa è tecnica di inferenza utilizzata in sistemi basati su regole. Partendo da un di fatti (le condizioni iniziali) e applicare ripetutamente regole di inferenza per dedurre nuovi fatti, finché non si raggiunge un obiettivo preposto. Un motore inferenziale è un algoritmo che simula le modalità con cui la mente umana trae delle conclusioni logiche attraverso il ragionamento.

1. Si parte dai fatti noti espressi nel secondo campo del check_fwd/3.

2. Si controlla se le le regole che sono già presenti al momento già permettono di arrivare al goal finale.

3. Si leggono le regole e le si verifica rispetto alle precondizioni date.

4. Si allarga l'insieme delle proposizioni vere, aggiungendo eventualmente quelle presenti all'interno delle implicazioni.

5. Si ricomincia dal punto 2

Strutture dati. Vengono utilizzate diverse strutture dati differenti la cui base sono le liste implementate nel linguaggio `Fabula`.

1. Liste: Per rappresentare insiemi di implicazioni, condizioni iniziali e finali.

2. Operatori logici (`and`,`or`): Rappresentati come alberi binari di espressioni logiche.

3. Implicazioni `impl(A, B)`: Strutture che rappresentano regole logiche.

4. Le implicazioni che non possono ancora essere verificate, per la mancanza della veridicità di uno o più antecedenti, vengono inserite in una lista con regole di gestione FIFO, simile a quello che accade in uno Stack.

Le implicazioni (`impl(A, B)`) possono contenere uno o più `and()` annidati, in modo simile il goal finale potrebbe essere rappresentato da uno o più proposizioni dentro ad uno o più `or()` annidati.

Funzioni. All'utente finale verrà esposta soltanto una funzione check/3, al cui interno inserirà nei rispettivi campi

1. La lista delle implicazioni sfruttando il termine impl(A,B).

2. La lista delle regole iniziali, rappresentata da una lista di proposizione, da cui partire per effettuare la computazione.

3. la lista delle regole finali che rappresentano l'obiettivo da raggiungere.

```
1 ▷ check/3 function
2 check(ListaImpl, RegoleIniziali, Goal)
3     if check_fwd(ListaImpl,[], RegoleIniziali, Goal)
```

L'implementazione del ragionatore tramite il "Forward chaining" avviene tramite la funzione check_fwd/3, la signature è simile alla check tranne per il secondo elemento che rappresenta lo Stack delle regole che si sono rivelate false fino a quel momento.

```
1 ▷ check_fwd/3 function pravate
2 check_fwd(__,_, X,Y) if anyMember(Y, X)
```

I 2 casi in cui la funziona termina con successo sono quelli in cui almeno un termine del goal è presente all'interno delle proposizioni verificate come vere.

```
1
2 ▷ check_fwd/3 function first case
3 check_fwd([impl(ListaAnd,C
4           StackImplicazioniFalse,
5           RegoleIniziali,
6           RegoleFinali) =
7     check_fwd(StackImplicazioniFalse,
```

```
8                 [],
9                 append(RegoleIniziali,
10                [normalizeNot(C)]),
11                RegoleFinali)
12    if subset(andList(ListaAnd),
13                RegoleIniziali)
```

Nel caso in cui sia presente soltanto una implicazione la si verifica, si normalizza la tesi se necessario e in caso positivo si amplia la lista delle proposizioni vere. Se la verifica è positiva si spostano le eventuali regole reputate false e le si inserisce all'interno delle regole da verificare.

```
1  ▷ check_fwd/3 function second case
2  check_fwd([impl(ListaAnd, C) | T],
3            StackImplicazioniFalse,
4            RegoleIniziali,
5            RegoleFinali) =
6     check_fwd([StackImplicazioniFalse | T],
7               [], append(RegoleIniziali,
8               [normalizeNot(C)]), RegoleFinali)
9     if subset(andList(ListaAnd),
10              RegoleIniziali)
```

Se la lista delle implicazioni è composta ancora da più di un elemento si verifica la prima e nel caso si amplia la lista delle regole vere e si accoda il resto delle regole alla coda "First in First out" delle regole reputate come false in precedenza. Questo fa si che la prossima computazione provi a verificare prima le regole che in precedenza non avevano tutti i propri antecedenti rispettati.

```
1  ▷ check_fwd/3 function third case
2  check_fwd([H | T],
3            StackImplicazioniFalse,
4            RegoleIniziali,
5            RegoleFinali)
6     if check_fwd(T, [H | StackImplicazioniFalse],
7                  RegoleIniziali,
8                  RegoleFinali)
```

Come ultimo caso se la implicazione non rientra in nessuno dei casi precedenti allora la si inserisce nella lista delle regole false che funziona come una coda FIFO. Questa lista è presente nel secondo argomento di check_fwd.

Per poter scrivere il modo più semplice le funzioni si poò effettuare l'ingovernabile dell'operatore `*` per sfruttarlo al posto degli `and()`.

```
1  '*'(X, Y) = and(X,Y)
2  '*'(X, and(Y,Z)) = and(X,Y)*Z
```

Esempi. Descrizione di alcuni esempi di utilizzo della funzione check.

Vengono presi in esame diverse combinazioni di liste di implicazioni, regole da cui partire e goal. Questa breve lista serve a esplicitare in quali casi il check riesca ad

Check	**Risultato**
`check([impl(and(a, c), d), impl(and(a, b), c)], [a, b], [d])`[1]	true
`check([impl(and(a, b), c), impl(and(a, c), d)], [a, b], [d])`[2]	true
`check([impl(and(a, b), c)], [a, b], [d])`[3]	NMR
`check([impl(and(a, b), c), impl(and(a, b), c)], [a, b], [d])`[4]	NMR
`check([],[a, b],[a])`[5]	true
`check([],[a, b, c],[a, d])`[6]	true
`check([impl(and(and(a, b), c), d)], [a, b, c], [d])`[7]	true
`check([impl(and(and(a,b),c),d),impl(and(c,d),e)],[a,b,c],[e])`[8]	true

Tabella 1: Risultati dei test esercizio 1

applicare le regole ed arrivare all'obiettivo preposto. Quando si ottiene un NMR[3,4] ("no more results") significa che l'interprete del linguaggio non è riuscito ad arrivare ad un risultato positivo e quindi non è stato possibile trovare almeno un goal appartenente all'insieme dei fatti ampliati. Il primo esempio[1] mostra come il ragionatore sia capace di arrivare al goal preposto (d) anche se le regole non sono ordinate, questo è possibile grazie allo stack delle implicazioni che inizialmente sono risultate come non verificabili. Quinto[5] e sesto[6] esempio sono i casi in cui il check è già verificato senza dover fare alcun passo e quindi daranno subito `true`.

Esercizio 2

Problema. In questo esercizio viene analizzato il problema della pianificazione, partendo da uno stato iniziale del mondo e cercando di arrivare a uno stato finale desiderato, definito come stato Goal. Ogni azione è caratterizzata da:

1. Nome univoco
2. Precondizione (sottoinsieme dello stato del mondo)
3. Postcondizione (sottoinsieme dello stato del mondo risultante dopo l'azione)
    - Quelli che vengono rimossi dallo stato del mondo (post condizioni delete)
    - Quelli che vengono aggiunti allo stato del mondo (post condizioni add)

Esempi di azioni:

1. `grab(X)`: afferra il blocco X
2. `put(X,Y)`: posiziona il blocco X su Y
3. `putOnTable(X)`: posiziona il blocco X sul tavolo

In seguito andrò ad elencare le azioni che possono essere intraprese con le relative precondizioni e post condizioni da applicare alla lista degli stati.

1. `grab(X)`:
    - **Precondizioni:** `empty, clear(X)`
    - **Post condizioni delete:** `empty`
    - **Post condizioni add:** `holding(X)`
2. `put(X,Y)`:
    - **Precondizioni:** `holding(X), clear(Y)`
    - **Post condizioni delete:** `holding(X), clear(Y)`
    - **Post condizioni add:** `empty, clear(X), on(X,Y)`
3. `putOnTable(X)`:
    - **Precondizioni:** `holding(X)`
    - **Post condizioni delete:** `holding(X)`
    - **Post condizioni add:** `empty, clear(X), onTable(X)`

Stati: I predicati per descrivere gli stati sono rappresentati come liste e includono le seguenti descrizioni:

- `on(X,Y)`: il blocco X è appoggiato su Y
- `onTable(X)`: il blocco X è sul tavolo
- `empty`: il braccio è libero

- `holding(X)`: il braccio ha in mano il blocco X
- `clear(X)`: il blocco X è libero e non ha nulla sopra

Gli stati verranno inseriti all'interno della rappresentazione corrente e futura del mondo che poi dovranno essere elaborati dal pianificatore per poter applicare le azioni ed avere il goal a cui puntare.

Esempio di stato iniziale:

[empty, on(b3,b1), onTable(b1), onTable(b2), clear(b3), clear(b2)]

Esempio di stato finale:

[empty, on(b1,b2), on(b2,b3), onTable(b3), clear(b1)]

La funzione block permette di dichiarare quali e quanti sono i blocchi sono presenti sul tavolo. La funzione `block(X)` può essere definita come segue:

- `block(X)` if `X = b1`
- `block(X)` if `X = b2`
- `block(X)` if `X = b3`

oppure in forma semplificata:

- `block(b1)`
- `block(b2)`
- `block(b3)`

Se per esempio si prova a controllare quali blocchi sono presenti sul tavolo tramite la query `[block(X), X]` il risultato sarà `[true,b1], [true,b2], [true,b3]`.

Metodo. Il metodo sviluppato per risolvere il problema del pianificatore si basa su un approccio logico-funzionale, implementato all'interno del linguaggio `Fabula`. Il sistema utilizza il metodo della forward chaining per generare piani a partire da un insieme di regole e predicati.

Sono state definite delle regole per le operazioni di base, come ad esempio `grab` e `put`, le quali modellano le azioni eseguibili nel dominio dei blocchi. Queste azioni sono rappresentate come predicati logici che il pianificatore utilizza per costruire un percorso di azioni che portano dallo stato iniziale a uno stato obiettivo. Ogni stato è descritto da una serie di predicati, tra cui `block`, `subset` e `removeElement`, che specificano la relazione tra i blocchi e il loro posizionamento. Quando una regola si applica, vengono eseguite le operazioni necessarie per aggiornare lo stato corrente. Il pianificatore ha come primo caso il controllo dello stato finale della computazione che potrebbe essere terminata nel caso in cui essa sia un sottoinsieme dello stato corrente. Durate l'applicazione della forward chaining il pianificatore controlla che esista almeno una regola da poter applicare le cui pre condizioni sono compatibili con lo stato corrente della computazione, dopo aver applicato le post condizioni passa alla fase successiva per trovare la futura azione da applicare.

Il problema è stato approssimato alla ricerca di un percorso all'interno di un grafo. Esistono diverse strategie per scegliere quale sia il migliore percorso da seguire per poter arrivare all'obiettivo del goal. La ricerca in profondità prevede di sfruttare la ricorsione per allontanarsi velocemente dalla radice dell'albero di decisione, ha il vantaggio di essere più efficiente a livello spaziale, ma al tempo stesso i piani che vengono generati non sono minimi. La ricerca in ampiezza al contrario sviluppa l'albero di decisione un livello alla volta, dando in output il percorso minimo per arrivare al goal, ma con lo svantaggio di dover mantenere in memoria i nodi che sono già stati attraversati. Una tecnica che riesce a sfruttare le caratteristiche di entrambe è la ricerca iterativa in profondità, in cui viene effettuata una DFS imponendo un limite di profondità massima, se non si è arrivati ad un piano completo si itera la ricerca, aumentando il limite.

Strutture dati. La base del pianificatore sono le azioni, funzioni che hanno il compito di aggiornare lo stato corrente della computazione sfruttando quelle che sono le necessarie pre e post condizioni. Le funzioni hanno arietà 3 e hanno danno come input i blocchi da cui da cui partire, lo stato Iniziale prima della computazione, lo stato finale dopo l'esecuzione dell'azione e il piano corrente. L'output sarà il piano aggiornato tramite una `append/2`. Le azioni accettano come input lo stato del mondo corrente e quello finale prefissato, in entrambi i casi si tratta di una lista contenente i predicati descritti in precedenza. Il Piano a sua volta sarà sempre una lista di stati in cui però verranno inserite le azioni effettuate dal pianificatore per arrivare allo stato finale, il piano è frutto delle pre e post condizioni presenti nelle regole.

Per velocizzare la computazioni si possono avere strutture di supporto come degli stack che contengono gli stati già visitati in precedenza per non incorrere in dei loop infiniti.

Funzioni. All'utente finale verranno esposte 2 funzioni trace/3 e trace_limit/4. La prima funzione pubblica andrà ad applicare in moto iterativo le varie azioni finché non si otterrà un risultato che verrà dato sotto forma di piano. La signature della funzione prevede di inserire lo stato di partenza, lo stato finale a cui si vuole arrivare e il piano che successivamente dovrà essere letto. La computazione si fermerà quando lo stato corrente sarà un sottoinsieme dello stato finale.

```
1 ▷ trace/3 function first case
2 trace(Stato, StatoFinale, Piano) = Piano
3     if subset(StatoFinale, Stato)
4 ▷ trace/3 function second case
5 trace(Stato, StatoFinale, Piano) = Piano2
6     if Piano1 = try_action(Stato, StatoNuovo, Piano),
7        Piano2 = trace(StatoNuovo, StatoFinale, Piano1)
```

Dato la natura esponenziale della computazione è stata idea anche la seconda funzione che introduce un quarto input che consiste in un contatore che forza il programma a cercare una soluzione con soltanto un numero prestabilito di passi.

```
1 ▷ trace_limit/3 function first case
2 trace_limit(Stato, StatoFinale, Piano, N) = Piano
3     if subset(StatoFinale, Stato)
4
```

```
▷ trace_limit/3 function first case
trace_limit(Stato, StatoFinale, Piano, N) = Piano2
    if N > 0,
       Piano1 = try_action(Stato, StatoNuovo, Piano),
       Piano2 = trace_limit(StatoNuovo,
                            StatoFinale,
                            Piano1, N-1)
```

Una ricerca iterativa in profondità può essere ottenuta tramite le seguenti funzioni, bisognerà partire da un limite iniziale, per esempio 0.

```
▷ planIter function
planIter(StatoIniziale, StatoFinale, N) =
    try_plan(StatoIniziale, StatoFinale, N)

▷ planIter private function first case
try_plan(StatoIniziale, StatoFinale, N) =
    trace_limit(StatoIniziale, StatoFinale, [], N)

    ▷ planIter private function second case
try_plan(StatoIniziale, StatoFinale, N) =
    planIter(StatoIniziale, StatoFinale, N + 1)
```

Le funzioni private prevedono le regole da applicare nella computazione e una funzione di supporto chiamata try_action il cui scopo è quello di provare iterativamente ogni regola. Gli input saranno lo stato iniziale, lo stato finale/obiettivo e il piano corrente.

```
▷ try_action private function first case
try_action(StatoIniziale, StatoFinale, Plan)
    = putOnTable(X, StatoIniziale, StatoFinale, Plan)

▷ try_action private function second case
try_action(StatoIniziale, StatoFinale, Plan)
    = grab(X, StatoIniziale, StatoFinale, Plan)

▷ try_action private function third case
try_action(StatoIniziale, StatoFinale, Plan)
    = put(X, Y, StatoIniziale, StatoFinale, Plan)

▷ try_action private function fourth case
trace_limit(Stato, StatoFinale, Piano, N) = Piano
    if subset(StatoFinale, Stato)
```

Successivamente propongo una implementazione delle funzioni che hanno come obiettivo l'implementare le 3 azioni necessarie per poter usare il pianificatore. Queste funzioni modellano le azioni di afferrare, posizionare su un altro blocco e posizionare sul tavolo. Ciascuna funzione opera prendendo uno stato iniziale, uno stato finale e un piano d'azione da aggiornare, eseguendo le modifiche necessarie allo stato e generando una sequenza di azioni da seguire.

```
▷ grab/4 function private function
```

```
grab(X, StatoIniziale, StatoFinale, Plan) =
    append(Plan, [grab(Stato)])
   if block(X),
    Stato = X,
    subset([empty, clear(X)],StatoIniziale),
    StatoFinaleIntermedio = append(StatoIniziale,
                                   [holding(X)]),
    StatoFinale = removeElement(StatoFinaleIntermedio,
                                [empty])
```

La funzione `grab` rappresenta l'azione di afferrare un blocco X da uno stato iniziale. Le condizioni iniziali richiedono che X debba essere un blocco, lo stato iniziale debba contenere un tavolo vuoto e il blocco X debba essere libero da altri oggetti.

```
▷ put/5 function private function
put(X, Y, StatoIniziale, StatoFinale, Plan) =
    append(Plan, [put(Stato)])
    if block(X),
    block(Y),
    Y ≠ X,
    Stato = [X, Y],
    subset([holding(X), clear(Y)], StatoIniziale),
    StatoFinaleIntermedio = append(StatoIniziale,
                                   [on(X, Y),
                                   clear(X),
                                   empty]),
    StatoFinale = removeList(StatoFinaleIntermedio,
                             [holding(X),
                             clear(Y)])
```

La funzione `put` modella l'azione di posizionare il blocco X sopra il blocco Y. Le condizioni iniziali richiedono che sia X sia Y siano blocchi distinti e che Y sia libero da altri blocchi, mentre X deve essere già in mano.

```
▷ putOnTable/4 private function
putOnTable(X, StatoIniziale, StatoFinale, Plan) =
append(Plan, [putOnTable(Stato)])
    if block(X),
    Stato = X,
    subset([holding(X)], StatoIniziale),
    StatoFinaleIntermedio = append(StatoIniziale,
                                   [onTable(X),
                                   clear(X),
                                   empty]),
    StatoFinale = removeElement(StatoFinaleIntermedio,
                                [holding(X)])
```

La funzione `putOnTable` descrive l'azione di posizionare il blocco X sul tavolo. Le condizioni iniziali richiedono che X sia blocco e sia già in mano.

Esempi. Esempi di operazioni con il piano di esecuzione.

- **Operazione:** ‘trace_limit([clear(b1), onTable(b1), clear(b2), onTable(b2), empty], [on(b1, b2), empty], Piano, 10)‘

  risultato: ‘[grab(b1), put([b1, b2])]‘

- **Operazione:** ‘trace_limit([clear(b1), clear(b3), onTable(b1), clear(b2), onTable(b2), empty], [on(b1, b2),on(b3, b1), empty],Piano,5)‘

  risultato: ‘[grab(b1),put([b1,b2]),grab(b3),put([b3,b1])]‘

- **Operazione:** ‘trace_limit([clear(b1), clear(b3), onTable(b1), clear(b2), onTable(b2), empty], [on(b2, b3),on(b2, b1), empty],Piano,5)‘

  risultato: ‘Nessun risultato‘

- **Operazione:** ‘trace_limit([clear(b1),clear(b2),clear(b3),empty], [on(b2, b3),holding(b1)], Piano,5)‘

  risultato: ‘[grab(b1),putOnTable(b1),grab(b2),put([b2,b3]),grab(b1)]‘

Esercizio 3

Problema. Il problema consiste nella creazione di un interprete capace di comprendere e processare frasi in linguaggio naturale riguardanti la posizione di blocchi su un tavolo. Il programma raggiunge questo obiettivo attraverso un parser che converte la frase in un albero sintattico astratto (AST) e, successivamente, semplifica tale struttura nella sua forma logica finale. Questo processo avviene utilizzando regole di inferenza per identificare gli oggetti in base alle loro proprietà. Infine, sarà possibile analizzare l'AST e rispondere a domande poste dall'utente.

Metodo. La creazione dell'interprete, basato su un albero di sintassi astratta, richiede di suddividere il problema in due sotto-problemi principali:

- Un parser, definito tramite il predicato tell/2, capace di interpretare le frasi di input e generare un albero di sintassi astratta, dal quale derivare i fatti relativi al micromondo.

- Un ragionatore, in grado di utilizzare i fatti del micromondo per comporre frasi coerenti sulle informazioni in suo possesso.

Per semplificare lo sviluppo del parser, si utilizzerà un micromondo che segue la grammatica inglese. Un micromondo è un ambiente limitato, con regole ben definite; in questo caso, si adotterà l'ambiente dei blocchi, già noto dai precedenti esercizi.

L'output del parser sarà nel medesimo formato della base di conoscenza utilizzata nel ragionatore per la pianificazione dell'esercizio 2. In un possibile approfondimento successivo, si potrebbero integrare parser e pianificatore per consentire l'inserimento in linguaggio naturale delle informazioni necessarie a individuare un percorso nel mondo dei blocchi.

Strutture dati. La principale struttura dati utilizzata nell'esercizio è l'albero di sintassi astratta (AST in Inglese) che consiste in un albero binario gerarchico dove viene rappresentato il modo astratto la sintassi del testo che gli viene dato come input.

La struttura AST di `[the, red, block, is, on, b1]` può essere scritta formalmente come:

$$[\text{on}(\text{object}(\text{block}, [\mathit{attribute}(\mathit{color}, \mathit{red})]), \text{object}(b1, []))]$$

Questa struttura può essere rappresentata graficamente come un albero:

Funzioni. Le funzioni che vengono esposte all'utente sono tell e ask. La prima permette di partire da una frase per arrivare ad un insieme di fatti, la seconda partendo da un insieme di fatti e una domanda permette di ottenere una risposta rispetto al micro mondo che viene descritto dal programma.

Le 2 funzioni che verranno esposte all'utente saranno tell/2 e ask/2. In entrambe le funzioni la prima lista di inuit riguarda la i fatti del mondo che si conoscono fino a quel momento, la seconda conterrà una nuova frase che dovrà essere analizzata. Nel caso della della ask bisognerà dare in output nuovi fatti e nel caso della ask bisognerà rispondere con una frase composta da termini.

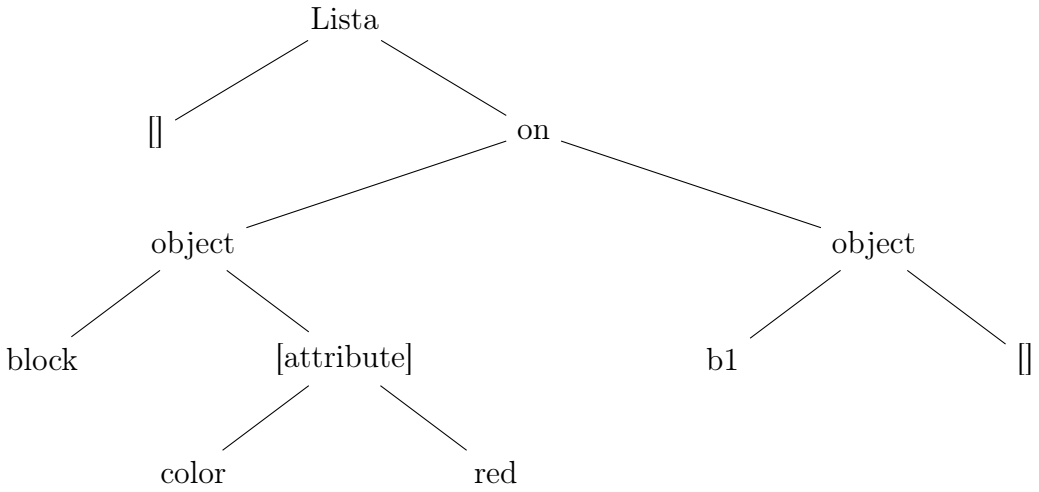

Figura 1: Rappresentazione ad albero della struttura AST per la relazione on()

```
1  tell(KnowledgeBase, Sentence) =
2      tell1(KnowledgeBase,Sentence)
3
4  ask(KnowledgeBase, Question) =
5      ask1(KnowledgeBase, Question)
```

Entrambe le funzioni riportano a 2 funzioni private che implementeranno le funzionalità delle ask e tell. Alla base delle 2 funzioni, come visto nel paragrafo relativa alle strutture dati utilizzate in questo esercizio, è presente una funzione che ha come obiettivo la creazione dell'albero di sintassi astratta basata sull'inglese.

```
1  ▷ AST ▷a([the, block,is, on, the, table])
2  a(A) = R
3      if np(A) = R1, vp(R1) = R
4  ▷ Noun Phrase
5  np(NP) = R
6      if det(NP) = R1, n(R1) = R
7  ▷ Verb Phrase
8  vp(VP) = R
9      if abe(VP) = R1, pp(R1) = R
10 ▷ Prepositional Phrase
11 pp(PP) = R
12     if PP = [on | R1], np(R1) = R
13 abe([B | R]) = R
14     if be(B)
15 ▷ Determiner
16 det([D | R]) = R
17     if determiner(D)
18 ▷ Noun Phrase
19 n([N | R]) = R
20     if noun(N)
21 n(N) = R
```

```
22      if N = [A, R1 | R], adj(A), noun(R1)
```

Questa prima versione ha come obiettivo quello di restituire come output la lista vuota, se viene riconosciuta tutta la frase, oppure una lista con all'interno la porzione della frase rimanente che non è stata riconosciuta nel processo di analisi.

```
1  ▷ AST a([the, block,is, on, the, table])
2  a(A) = vp(R1, ASTN)
3      if [R1, ASTN] = np(A)
4  c(C) = vp
5  ▷ Noun Phrase
6  np([the | R1]) = n(R1)
7  ▷ Verb Phrase
8  vp([is, on | N], ASTS) = [R, on(ASTS, ASTN)]
9      if [R, ASTN] = np(N)
10 ▷ Noun Phrase
11 n([N | R]) = [R, object(C, [])]
12     if noun(C, N)
13 n(N) = [R, object(C, [attribute(T, A)])]
14     if N = [A, L | R], adj(T, A), noun(C, L)
```

Questa seconda versione ha come output l'albero di sintassi astratta con anche la possibilità di inserire aggettivi come colori e dimensioni. Tali aggettivi devono essere aggiunti tramite predicati come.

```
1  ▷ adj/2
2  adj(color, yellow)
3  adj(color, blue)
4  adj(color, red)
5  adj(size, small)
6  adj(size, large)
```

La tell/2 accetterà frasi del tipo:

$$[[\texttt{b1}, \texttt{is}, \texttt{on}, \texttt{the}, \texttt{table}, \texttt{'.'}]]$$

Queste frasi dovranno passare da un processo di pre elaborazione per eliminare il punto finale ed essere elaborate dalla funzione che crea l'albero di sintassi astratta

```
1  ▷ tell1
2  tell1(KnowledgeBase, Sentence) = simplify_ast(AST)
3      if SentenceWithoutPeriod = remove_period(Sentence),
4         AST = a(SentenceWithoutPeriod)
```

La funzione simplify_ast/1 prende in input l'albero di sintassi astratta e produce l'ouput che verrà mostrato dalla funzione tell1/2. Per esempio nel caso della precedente esempio sarà sufficiente una funzione come la seguente.

```
1  ▷simplify_ast/1
2  simplify_ast([[],on(object(X,[]),object(table,[]))]) =
3      [onTable(X)]
```

Se gli input presentano degli aggettivi come i colori dei blocchi e le loro dimensioni si possono usare più casi della funzione simplify_ast/1 per catturare ogni evenienza.

```
▷simplify_ast/1
simplify_ast([[],
              on(object(Obj,[attribute(color,COLOR)]),
              object(table,[]))]) =
        [onTable(Blocco), color(Blocco, COLOR)]
        if blocchi_generici(Obj),
            blocks(Blocco),
            color(Blocco) = COLOR

▷simplify_ast/1
simplify_ast([[],
              on(object(Obj,[attribute(size,SIZE)]),
              object(table,[]))]) =
     [onTable(Blocco), size(Blocco, SIZE)]
     if blocchi_generici(Obj),
     blocks(Blocco), size(Blocco) = SIZE
```

Le versioni più semplici della funzione ask hanno il computo di cercare all'interno della base di conoscenza le informazioni necessarie per finire le frasi.

```
▷ask1/2
ask(KnowledgeBase, [where, is, Object, '?']) =
    [Object, is, on, the, table, '.']
    if member(KnowledgeBase, onTable(Object))

▷ask1/2
ask(KnowledgeBase, [what, color, is, Object, '?']) =
    [Object, is, Color, '.']
    if member(KnowledgeBase, color(Object, Color))

▷ask1/2
ask(KnowledgeBase, [where, is, Object2, '?']) =
    [Object2, is, on, Object1, '.']
    if member(KnowledgeBase, on(Object1, Object2)),
    Object1 ≠ Object2
```

Questa base di partenza permette di scrivere predicati in cui tell e ask sono annidati una dentro l'altra, la prima fornisce alla seconda le informazioni necessarie per scrivere le risposte.

$$\text{ask}(\text{tell}([], [\texttt{b1}, \texttt{is}, \texttt{on}, \texttt{the}, \texttt{table}, \texttt{'.'}]), [\texttt{where}, \texttt{is}, \texttt{b1}, \texttt{'?'}]) = [\texttt{b1}, \texttt{is}, \texttt{on}, \texttt{the}, \texttt{table}, \texttt{'.'}]$$

simplyfy_ ast e *ask* può essere estesa per poter tenere in considerazione di più casi, per esempio si può specificare nella base di conoscenza il posizionamento dei blocchi e una loro caratteristica come colore o dimensione per permettere al processo non deterministico di determinare di che blocchi si sta parlando.

```
▷ Funzione principale ask estesa
ask(KnowledgeBase, [where, is, the, Adj, block, '?']) =
    [Block, is, on, the, table, '.']
    if adj(Category, Adj),
       Block =
       find_block_with_property(KnowledgeBase, Category, Adj),
       member(KnowledgeBase, onTable(Block))

ask(KnowledgeBase, [where, is, the, Adj, Noun, '?']) =
    [Block, is, on, OtherBlock, '.']
    if adj(Category, Adj),
       Block =
       find_block_with_property(KnowledgeBase, Category, Adj),
       member(KnowledgeBase, on(Block, OtherBlock))
```

In questo caso vengono utilizzate funzioni di supporto che possano far coincodere la categoria con l'oggetto.

```
▷ find_block_with_property/3
find_block_with_property(KnowledgeBase, size, Size) =
    Block
    if member(KnowledgeBase, size(Block, Size))

find_block_with_property(KnowledgeBase, color, Color, Block) =
    Block
    if member(KnowledgeBase, color(Block, Color))
```

Il caso con *over* e *on top of* è simile a on, bisogna interpretare l'albero sintattico astratto e prevedere la sua gestione.

```
a([the | Rest]) = vp(R1, ASTN)
    if [R1, ASTN] = np1([the | Rest])

np1([the, Color, block | Rest]) =
    [Rest, object(B, [attribute(color, Color)])]
    if adj(color, Color),
       blocks(B),
       color(B) = Color

vp([is, red], object(B, Attrs))
  = [[], color(object(B, Attrs), red)]

vp([is, on, the, top, of | Rest], Obj) = [R, on(Obj, Target)]
    if [R, Target] = np2(Rest)

vp([is, on | Rest], Obj) = [R, on(Obj, Target)]
    if [R, Target] = np2(Rest)

vp([that,is, on | Rest], Obj) = [R, on(Obj, Target)]
```

```
20      if [R, Target] = np2(Rest)
21
22  vp([is, over | Rest], Obj) = [R, over(Obj, Target)]
23      if [R, Target] = np2(Rest)
24
25  vp([is, below | Rest], Obj) = [R, below(Obj, Target)]
26      if [R, Target] = np2(Rest)
27
28  np2([the, table]) = [[], object(table, [])]
29
30  np2([the, Size, block | Rest]) =
31      [Rest, object(B, [attribute(size, Size)])]
32      if adj(size, Size),
33         blocks(B),
34         size(B) = Size
```

Modificando l'analisi dell'albero AST si possono leggere frasi più complesse come "the, blue, block, is, on, the, large, block".

Esempi. Esempi di operazioni con la base di conoscenza

- **Operazione:** ‘a([b1, is, on, the, table])‘
  risultato: ‘[[], on(object(b1, []), object(table, []))]‘
- **Operazione:** ‘a([b1, is, on, b2])‘
  risultato: ‘[[],on(object(b1,[]),object(b2,[]))]‘
- **Operazione:** ‘a([the, blue, block, is, on, b2])‘
  risultato: ‘[[],on(object(block,[attribute(color,blue)]),object(b2,[]))]‘
- **Operazione:** ‘ask([onTable(b1)], [where, is, b1, ’?’])‘
  risultato: ‘[b1, is, on, the, table, ’.’]‘
- **Operazione:** ‘ask([on(b1, b2)], [where, is, b2, ’?’])‘
  risultato: ‘[b2, is, on, b1, ’.’]‘
- **Operazione:** ‘tell([], [b1, is, on, the, table, ’.’])‘
  risultato: ‘[onTable(b1)]‘
- **Operazione:** ‘ask(tell([], [b1, is, on, the, table, ’.’]), [where, is, b1, ’?’])‘
  risultato: ‘[b1, is, on, the, table, ’.’]‘
- **Operazione:** ‘ask([on(b1,b2), size(b1, small)], [where, is, the, small, block, ’?’])‘
  risultato: ‘[b1,is,on,b2,’.’]‘
- **Operazione:** ‘a([the, blue, block, is, over, the, table])‘
  risultato: ‘[[],over(object(b2,[attribute(color,blue)]),object(table,[]))]‘
- **Operazione:** ‘tell([], [the, blue, block, is, over, the, table, ’.’])‘
  risultato: ‘[onTable(b2),color(b2,blue)]‘

- **Operazione:** ‘a([the, blue, block, is, below, the, table])‘
  risultato: ‘[[],below(object(b2,[attribute(color,blue)]),object(table,[]))]‘
- **Operazione:** ‘a([the, blue, block, is, on, the, top, of, the, large, block])‘
  risultato: ‘[[],on(object(b2,[attribute(color,blue)]),object(b3,[attribute(size,large)]))]‘
- **Operazione:** ‘tell([on(b3,b2)],[the, blue, block, is, on, the, top, of, the, large, block, ’.’])‘
  risultato: ‘Nessun risultato possibile ‘

Esercizio 4

Problema. L'obiettivo del problema è quello di creare un calcolatore simbolico in grado di manipolare espressioni matematiche simboliche, con particolare attenzione alle espressioni polinomiali multivariate e a funzionalità come somma, prodotto, derivazione.

Il problema può essere affrontato tramite una rappresentazione densa e sparsa. Successivamente si vedrà una prima rappresentazione compatta a variabile singola, sparsa a variabile singola e sparsa multivariata.

Polinomi in variabile singola

Rappresentazione:

- **Versione Densa**: Rappresentazione del polinomio tramite un array con coefficienti per ogni grado.
  Esempio: `poly(x, 4, [4, 7, 0, 0, 9])` rappresenta il polinomio $4 + 7x + 9x^4$.
- **Versione Sparsa**: Rappresentazione dei soli gradi con coefficienti diversi da zero.
  Esempio: `poly(x, 4, [k(0, 4), k(1, 7), k(4, 9)])` rappresenta $4 + 7x + 9x^4$.

La prima implementazione è stata effettuata tramite il metodo denso dala la sua semplicità di implementazione grazie alla manipolazione delle liste offerta dai predicati implementabili nel linguaggio.

Operazioni da supportare nel calcolatore:

- **Somma e Prodotto**: Definite simbolicamente con $+'(X, Y)$ e $*'(X, Y)$.
- **Derivazione**: Calcolo della derivata rispetto a una variabile: $\text{diff}(X, x)$.

Per consentire al calcolatore simbolico di interpretare correttamente le espressioni simboliche inserite dall'utente, è necessario implementare un parser. Questo parser traduce i termini in input in un formato comprensibile dal calcolatore. L'implementazione prevede la riscrittura degli operatori `+`, `*` e `^` per gestire tutti i casi in cui è necessario interagire con il predicato `poly`.

Polinomi in più variabili

Supporto:

- Polinomi con variabili multiple (es., x, y, z).
- Ordinamento delle variabili per la manipolazione: `ord([x, y, z])` specifica l'ordine di priorità.

Esempi di rappresentazione:

- $x^3 + 5y - 1$ è rappresentato come:

$$\text{poly}(x, [-1, \text{poly}(y, [0, 5]), 0, 1])$$

- $z^4 + 7x + y$ è rappresentato come:

$$\text{poly}(x, [\text{poly}(y, [\text{poly}(z, [0, 0, 0, 0, 1]), 1]), 7])$$

Metodo. La prima versione del problema vede la risoluzione delle operazioni base da effettuare sui polinomi con la rappresentazione densa a singola variabile. Tale rappresentazione permette di effettuare con maggiore facilità operazioni di somma e prodotto grazie alla manipolazione delle liste, per facilitare l'approccio il grado viene mantenuto esplicito, successivamente potrebbe essere dedotto tramite funzioni come `size`.

Per effettuare la somma è sufficiente calcolare quale sia il grado maggiore tra i 2 polinomi, estendere le loro liste se necessario ed effettuare la somma grado per grado.

Nel prodotto il grado del polinomio risultante è dato dalla somma dei gradi dei due polinomi originali. Questo determina la lunghezza della lista dei coefficienti del risultato. Si crea una lista di zeri di lunghezza pari al grado massimo risultante più uno, rappresenta i coefficienti del polinomio risultante. Si itera su tutti i termini del primo polinomio e, per ciascuno di essi, si moltiplica il coefficiente per ciascun termine del secondo polinomio. Il prodotto ottenuto viene aggiunto al termine corrispondente nella lista dei coefficienti del risultato, posizionato all'indice corrispondente alla somma degli esponenti dei due termini moltiplicati. Questo processo viene ripetuto per tutti i termini del primo polinomio e del secondo polinomio, accumulando i valori dei coefficienti nella lista del risultato.

Per la derivata è sufficiente iterare sui coefficienti e diminuire il grado del polinomio.

Successivamente sono passato a una rappresentazione densa. Le operazioni sono simili, ma invece di utilizzare una lista per rappresentare i coefficienti del polinomio, sfrutto un predicato $k(X, Y)$, dove X rappresenta il grado del polinomio e Y il valore del coefficiente corrispondente a quel grado. Una possibile variante potrebbe essere l'utilizzo di liste ricorsive al posto del predicato indicato.

L'ultima versione del programma vede l'utilizzo delle operazioni di somma, prodotto e derivazione su una versione densa dei polinomi in più variabili.

Strutture dati. La struttura dati principale che viene utilizzata all'interno del programma è la rappresentazione dei polinomi tramite il termine poly/2. Nella versione finale del programma verrà utilizzato il termine rat/2 per poter manipolare i numeri razionali all'interno delle operazioni base.

Funzioni. All'utente finale vengono esposte funzioni il cui unico scopo è sovrascrivere gli operatori di default messi a disposizione da Fabula per la somma e il prodotto. L'idea è che l'utente possa le operazioni di base messe a disposizione come se fossero operazioni built-in del linguaggio. L'unico nuovo predicato esposto sarà quello dedicato alla derivazione, `diff/1`. Tutte le le funzioni qui rappresentate, se non espressamente detto, sono da considerare come funzioni private. Successivamente vado a riportare le 3 implementazioni, ognuna presenterà in ordine gli quali operatori sono stati sovrascritti, la funzione per implementare la derivazione, le funzioni di somma/prodotto e se presenti ulteriori funzioni di supporto.

Densa ad una variabile Sovrascrittura degli operatori

Funzione per implementare l'operazione della derivazione

```
1 diff(poly(x, Grado, Lista)) =
2     poly(x, Grado - 1, ListaDerivata) if
3         ListaDerivata = diffList(Lista, 0)
4
5 diffList([], _) = []
```

```
diffList([H | T], Exp) =
    diffList(T, Exp + 1) if Exp = 0
diffList([H | T], Exp) =
    [H * Exp | diffList(T, Exp + 1)] if Exp > 0
```

Funzioni private per l'implementazione della somma e prodotto.

```

sum(poly(x, GradoX, ListaX), poly(x, GradoY, ListaY)) =
    poly(x, MaxGrado, SommaListe) if
        MaxGrado = max(GradoX, GradoY),
        ListaXEstesa = extend(ListaX, MaxGrado + 1),
        ListaYEstesa = extend(ListaY, MaxGrado + 1),
        SommaListe = sumLists(ListaXEstesa, ListaYEstesa)

extend(Lista, Len) = Lista if length(Lista) = Len
extend(Lista, Len) = extend([0 | Lista], Len)
                    if length(Lista) < Len


prod(poly(x, GradoX, ListaX), poly(x, GradoY, ListaY)) =
    poly(x, GradoX + GradoY, ProdottoListe) if
        LunghezzaRisultato = GradoX + GradoY + 1,
        ListaZeri = createZeros(LunghezzaRisultato),
        ProdottoListe =
            prodLists(ListaX, 0, ListaY, ListaZeri)

createZeros(0) = []
createZeros(N) = [0 | createZeros(N - 1)] if N > 0

prodLists([], _, _, Acc) = Acc
prodLists([H1 | T1], Exp1, Lista2, Acc) =
    prodLists(T1, Exp1 + 1, Lista2, NuovoAcc) if
        TempAcc = prodSingleTerm(H1, Exp1, Lista2, 0, Acc),
        NuovoAcc = TempAcc

prodSingleTerm(_, _, [], _, Acc) = Acc
prodSingleTerm(Coeff1, Exp1, [H2 | T2], Exp2, Acc) =
    prodSingleTerm(Coeff1, Exp1, T2, Exp2 + 1, NuovoAcc) if
        Prodotto = Coeff1 * H2,
        ExpTotale = Exp1 + Exp2,
        NuovoAcc = updateAtIndex(Acc, ExpTotale, Prodotto)
```

Funzioni di supporto

```
updateAtIndex([], _, _) = []
updateAtIndex([H | T], 0, Val) = [H + Val | T]
updateAtIndex([H | T], Idx, Val) =
    [H | updateAtIndex(T, Idx - 1, Val)] if Idx > 0

```

```
sumLists([], []) = []
sumLists([H1 | T1], [H2 | T2]) =
    [H1 + H2 | sumLists(T1, T2)]

max(X, Y) = X if X > Y
max(X, Y) = Y if Y > X
max(X, Y) = X if X = Y

isPoly(X) = true if X = poly(Var, Grad, Lista)
```

Sparsa ad una variabile Per poter utilizzare gli operatori '+' e '*' all'interno della console di fabula nel caso in cui si vogliano effettuare prodotti e somme tra polinomi bisogna effettuarne la sovrascrittura.

```
'^'(X, Y) = poly(x, Y, [k(Y, 1)]) if X = x, integer(Y)

'+'(x, Number) = poly(x, 0, [k(0, Number)])
    if integer(Number)
'+'(Number, x) = poly(x, 0, [k(0, Number)])
    if integer(Number)

▷ Aggiungere una regola per sommare
▷ un polinomio con un intero
'+'(poly(Var, Grado, Lista), Intero) =
    poly(Var, Grado, NuovaLista) if
        integer(Intero),
        NuovaLista = mergeSparse(Lista, [k(0, Intero)])

'+'(Intero, poly(Var, Grado, Lista)) =
    poly(Var, Grado, NuovaLista) if
        integer(Intero),
        NuovaLista = mergeSparse([k(0, Intero)], Lista)

'+'(X, Y) = Z if
    isPoly(X),
    isPoly(Y),
    Z = sum(X, Y)

'*'(X, Y) = Z if
    isPoly(X),
    integer(Y),
    Z = prod(X, poly(x, 0, [k(0, Y)]))

'*'(X, Y) = Z if
    isPoly(Y),
    integer(X),
    Z = prod(poly(x, 0, [k(0, X)]), Y)

```

```
'*'(X, Y) = Z if
    Z = prod(X, Y),
    isPoly(Z)
```

Grazie ad essere sarà possibile poter scrivere nel linguaggio naturale i polinomi con cui si vuole interagire ed in automatico verranno formati le strutture dati poly sfruttate all'interno delle funzioni.

Funzione per implementare l'operazione della derivazione

```
diff(poly(x, Grado, Lista)) =
    poly(x, Grado - 1, ListaDerivata) if
        ListaDerivata = diffList(Lista)

diffList([]) = []
diffList([k(0,_) | T]) = diffList(T)
diffList([k(E,C) | T]) =
    [k(E-1, C * E) | diffList(T)] if E > 0
```

Funzioni private per l'implementazione della somma e prodotto.

```
sum(poly(x, GradoX, ListaX), poly(x, GradoY, ListaY)) =
    poly(x, MaxGrado, SommaListe) if
        MaxGrado = max(GradoX, GradoY),
        SommaListe = mergeSparse(ListaX, ListaY)


mergeSparse([], Lista) = Lista
mergeSparse(Lista, []) = Lista
mergeSparse([k(E1,C1) | T1], [k(E2,C2) | T2]) =
    [k(E1,C1) | mergeSparse(T1, [k(E2,C2) | T2])] if E1 < E2
mergeSparse([k(E1,C1) | T1], [k(E2,C2) | T2]) =
    [k(E2,C2) | mergeSparse([k(E1,C1) | T1], T2)] if E2 < E1
mergeSparse([k(E,C1) | T1], [k(E,C2) | T2]) =
    [k(E,C1 + C2) | mergeSparse(T1, T2)]

prod(poly(x, GradoX, ListaX), poly(x, GradoY, ListaY)) =
    poly(x, GradoX + GradoY, ProdottoListe) if
        LunghezzaRisultato = GradoX + GradoY + 1,
        ListaZeri = createZerosK(LunghezzaRisultato, 0),
        ProdottoListe = prodLists(ListaX, ListaY, ListaZeri)



prodLists([], _, Acc) = Acc
prodLists([k(E1,C1) | T1], Lista2, Acc) =
    prodLists(T1, Lista2, NuovoAcc) if
        TempAcc = prodSingleTerm(k(E1,C1), Lista2, Acc),
        NuovoAcc = TempAcc

prodSingleTerm(_, [], Acc) = Acc
```

```
31 prodSingleTerm(k(E1,C1), [k(E2,C2) | T2], Acc) =
32     prodSingleTerm(k(E1,C1), T2, NuovoAcc) if
33         Prodotto = C1 * C2,
34         ExpTotale = E1 + E2,
35         NuovoAcc = updateAtIndexK(Acc, ExpTotale, Prodotto)
```

Le funzioni di supporto

```
1  createZerosK(0, _) = []
2  createZerosK(N, Curr) =
3      [k(Curr,0) | createZerosK(N - 1, Curr + 1)] if N > 0
4
5  updateAtIndexK([], _, _) = []
6  updateAtIndexK([k(E,C) | T], E, Val) = [k(E,C + Val) | T]
7  updateAtIndexK([k(E,C) | T], Idx, Val)
8      = [k(E,C) | updateAtIndexK(T, Idx, Val)] if E < Idx
9
10 max(X, Y) = X if X > Y
11 max(X, Y) = Y if Y > X
12 max(X, Y) = X if X = Y
```

Sparsa a più di una variabile La seguente soluzione va ad ovviare al problema dell'implementazione di una soluzione sparsa con più di una variabile. In questa rappresentazione non viene tenuta traccia del grado del polinomio, poly si esprime soltanto con 2 paramentri, variabile e lista di coeffieicnti *k()*. La chiave della soluzione sta nell'individuare un ordinamento assoluto tramite `ordinePrecedenza/2` e analizzare i diversi casi che vengono a presentarsi. La sovrascrittura degli operatori sarà simile a quella già vista nella versione sparsa ad 1 sola variabile. Il problema come visto successivamente sarà adattabile anche a poter elaborare numeri razionali.

```
1 ▷ Ordine di precedenza: x > y > z
2 ordinePrecedenza(x, y)
3 ordinePrecedenza(x, z)
4 ordinePrecedenza(y, z)
```

La funzione privata principale che deve gestire la somma dei polinomi ha 3 casi diffenreti:

1. Quando le 2 variabili sono uguali, per esempio si sommano 2 polinomi in x
2. quando la prima variabile precede la seconda
3. quando la seconda variabile precede la prima

Le seguenti funzioni private non verranno messe direttamente a disposizione del utente, ma bisgnerà passare dagli operatori + e *.

- `sumPoly`: somma di due polinomi,
- `mergeCoefficients`: unione di liste di coefficienti in ordine di esponente,
- `sumCoefficient`: somma di coefficienti (interi o polinomi),

- `productPoly`: prodotto di due polinomi,
- `productCoeffs`, `multiplyTermList` e `productCoefficient`: funzioni di supporto per calcolare il prodotto.

Le funzioni utilizzano una struttura dati del tipo `poly(Var, L)` dove:

- `Var` è la variabile (ad esempio `x`, `y` o simili),
- `L` è una lista di coppie `k(E, C)`, con:
  - `E`: esponente (intero),
  - `C`: coefficiente (che può essere un numero o un altro polinomio).

La funzione `sumPoly` somma due polinomi. Gestisce sia il caso in cui le due variabili esterne coincidono, sia il caso in cui sono diverse. In quest'ultimo caso, l'ordine di precedenza (`ordinePrecedenza(Var1, Var2)`) stabilisce quale variabile compare "più esternamente" nel polinomio risultante.

```
1  ▷ Caso 1: stessa variabile esterna
2  sumPoly(poly(Var, L1), poly(Var, L2)) = poly(Var, Merged)
3      if Var = Var,
4         Merged = mergeCoefficients(Var, L1, L2)
5
6  ▷ Caso 2: variabili esterne diverse, Var1 > Var2
7  sumPoly(poly(Var1, L1), poly(Var2, L2)) = poly(Var1, Merged)
8      if Var1 ≠ Var2,
9         ordinePrecedenza(Var1, Var2),
10        ▷ annido poly(Var2, L2) come coeff di grado 0
11        Merged =
12        mergeCoefficients(Var1, L1, [k(0, poly(Var2, L2))])
13
14 ▷ Caso 3: variabili esterne diverse, Var2 > Var1
15 sumPoly(poly(Var1, L1), poly(Var2, L2)) = poly(Var2, Merged)
16     if Var1 ≠ Var2,
17        ordinePrecedenza(Var2, Var1),
18        ▷ annido poly(Var1, L1) come coeff di grado 0
19        Merged =
20        mergeCoefficients(Var2, L2, [k(0, poly(Var1, L1))])
```

La funzione `mergeCoefficients` unisce due liste di termini (nella forma `[k(E, C)]`) in un'unica lista, ordinata per esponente crescente, sommando i coefficienti se l'esponente coincide.

```
1
2  ▷ sommando i coefficienti quando E coincide.
3  mergeCoefficients(_, [], L2) = L2
4  mergeCoefficients(_, L1, []) = L1
5
6  ▷ Caso E1 == E2 => sommiamo i coefficienti
7  mergeCoefficients(Var, [k(E, C1) | T1], [k(E, C2) | T2]) =
```

```
8       [k(E, C3) | T3]
9       if C3 = sumCoefficient(C1, C2),
10         T3 = mergeCoefficients(Var, T1, T2)
11
12  ▷ Caso E1 < E2
13  mergeCoefficients(Var, [k(E1, C1) | T1], [k(E2, C2) | T2]) =
14      [k(E1, C1) | T3]
15      if E1 < E2,
16         T3 = mergeCoefficients(Var, T1, [k(E2, C2) | T2])
17
18  ▷ Caso E2 < E1
19  mergeCoefficients(Var, [k(E1, C1) | T1], [k(E2, C2) | T2]) =
20      [k(E2, C2) | T3]
21      if E2 < E1,
22         T3 = mergeCoefficients(Var, [k(E1, C1) | T1], T2)
```

La funzione `sumCoefficient` gestisce la somma di due coefficienti, che possono essere sia interi sia polinomi (di qualunque variabile). La logica prevede:

```
1   ▷ Somma di due coefficienti
2   ▷ che possono essere numeri o polinomi (in qualunque Var).
3
4   ▷ Caso A: entrambi numeri
5   sumCoefficient(C1, C2) = S
6       if integer(C1),
7          integer(C2),
8          S = C1 + C2
9
10  ▷ Caso B: entrambi polinomi
11  sumCoefficient(C1, C2) = S
12      if isPoly(C1),
13         isPoly(C2),
14         S = sumPoly(C1, C2)
15
16  ▷ Caso C: C1 intero, C2 polinomio
17  sumCoefficient(C1, C2) = S
18      if integer(C1),
19         isPoly(C2),
20         VarC2 = polyVar(C2),
21         CoeffsC2 = polyCoeffs(C2),
22         ▷ Converto l'intero C1 in un
23         ▷ poly con la stessa var di C2
24         PolyC1 = poly(VarC2, [k(0, C1)]),
25         S = sumPoly(PolyC1, poly(VarC2, CoeffsC2))
26
27  ▷ Caso D: C2 intero, C1 polinomio
28  sumCoefficient(C1, C2) = S
29      if integer(C2),
30         isPoly(C1),
```

```
        VarC1 = polyVar(C1),
        CoeffsC1 = polyCoeffs(C1),
        ▷ Converto l'intero C2 in un poly
        ▷ con la stessa var di C1
        PolyC2 = poly(VarC1, [k(0, C2)]),
        S = sumPoly(poly(VarC1, CoeffsC1), PolyC2)
```

La funzione `productPoly` calcola il prodotto di due polinomi, facendo uso della funzione di supporto `productCoeffs` e, a cascata, di `multiplyTermList` e `productCoefficient`. Analogamente alla somma, se le variabili esterne differiscono si stabilisce chi va in testa in base all'`ordinePrecedenza`.

```
productPoly(poly(Var, L1), poly(Var, L2)) = poly(Var, L3)
    if L3 = productCoeffs(Var, L1, L2)

productPoly(poly(Var1, L1), poly(Var2, L2)) = poly(Var1, L3)
    if ordinePrecedenza(Var1, Var2),
       L3 = productCoeffs(Var1, L1, [k(0, poly(Var2, L2))])

productPoly(poly(Var1, L1), poly(Var2, L2)) = poly(Var2, L3)
    if ordinePrecedenza(Var2, Var1),
       L3 = productCoeffs(Var2, L2, [k(0, poly(Var1, L1))])
```

La funzione `productCoeffs` si occupa di moltiplicare liste di termini (cioè di coefficienti annotati con il rispettivo esponente). Per farlo, richiama la funzione `multiplyTermList` su ogni termine della prima lista, e poi utilizza `mergeCoefficients` per unire i risultati parziali.

```
productCoeffs(_, [], _) = []
productCoeffs(_, _, []) = []
productCoeffs(Var, [k(E1, C1)|T1], L2) = R
    if A = multiplyTermList(E1, C1, L2),
       B = productCoeffs(Var, T1, L2),
       R = mergeCoefficients(Var, A, B)

multiplyTermList(_, _, []) = []
multiplyTermList(E1, C1, [k(E2, C2)|T2]) = [k(E, D)|T]
    if E = E1 + E2,
       D = productCoefficient(C1, C2),
       T = multiplyTermList(E1, C1, T2)
```

La funzione `productCoefficient` gestisce invece il calcolo del prodotto tra due coefficienti (interi o polinomi).

```
productCoefficient(C1, C2) = S
    if integer(C1),
       integer(C2),
       S = C1 * C2

productCoefficient(C1, C2) = S
    if isPoly(C1),
```

```
        isPoly(C2),
        S = productPoly(C1, C2)

productCoefficient(C1, C2) = S
    if integer(C1),
        isPoly(C2),
        V2 = polyVar(C2),
        L2 = polyCoeffs(C2),
        P1 = poly(V2, [k(0, C1)]),
        S = productPoly(P1, poly(V2, L2))

productCoefficient(C1, C2) = S
    if integer(C2),
        isPoly(C1),
        V1 = polyVar(C1),
        L1 = polyCoeffs(C1),
        P2 = poly(V1, [k(0, C2)]),
        S = productPoly(poly(V1, L1), P2)
```

Le seguenti sono funzioni di supporto utilizzate all'interno del codice precedente. `polyVar` serve per estrarre la prima variabile da un polinomio, `polyCoeffs` in modo complementare estrae i coefficienti.

```

polyVar(poly(Var, _)) = Var

▷ Estrae la lista dei termini k(E, C)
polyCoeffs(poly(_, CoeffList)) = CoeffList
```

`diff` è la funzione pubblica che ha il compito di permettere di eseguire la prima derivata ad un polinomio multi variabile. `diff` permette di specificare in quale variabile si vuole effettuare la derivata e il polinomio da derivare. Se la variabile del polinomio coincide con quella di derivazione, ogni termine $\texttt{k}(E, C)$ con $E > 0$ viene trasformato in $\texttt{k}(E-1, E \times C)$, cioè si riduce l'esponente di uno e si moltiplica il coefficiente per l'esponente. Invece, se la variabile del polinomio è diversa da quella su cui si effettua la derivazione, il risultato è 0.

Per esempio consideriamo il polinomio in x:

$$\texttt{poly}(x, [\texttt{k}(0, 5),\ \texttt{k}(2, 2)]).$$

Che corrisponde a $5 + 2x^2$, se calcoliamo la derivata rispetto a x:

$$\frac{d}{dx}\left(5 + 2x^2\right) = 4x.$$

Nella tua rappresentazione:

- Il termine $\texttt{k}(0, 5)$ (5) scompare perché la derivata di una costante è 0.
- Il termine $\texttt{k}(2, 2)$ ($2x^2$) diventa $\texttt{k}(1, 4)$, poiché $E = 2$ e $C = 2$ danno $E - 1 = 1$ e $E \times C = 2 \times 2 = 4$.

```
diff(Var, poly(Var, Coeffs)) = poly(Var, Diffs)
    if Diffs = differentiateCoeffs(Var, Coeffs)

diff(Var, poly(Var2, _)) = 0
    if Var ≠ Var2

differentiateCoeffs(_, []) = []
differentiateCoeffs(Var, [k(E, C) | T]) = R
    if E > 0,
       C1 = scalarProduct(E, C),
       E1 = E - 1,
       T1 = differentiateCoeffs(Var, T),
       R = [k(E1, C1) | T1]
differentiateCoeffs(Var, [k(E, _) | T]) = R
    if E = 0,
       R = differentiateCoeffs(Var, T)

scalarProduct(_, 0) = 0
scalarProduct(N, C) = M
    if integer(C),
       M = N * C
scalarProduct(N, poly(V, Coeffs)) = poly(V, Coeffs1)
    if Coeffs1 = scalarProductCoeffs(N, Coeffs)

scalarProductCoeffs(_, []) = []
scalarProductCoeffs(N, [k(E, C) | T]) = [k(E, C1) | T1]
    if C1 = scalarProduct(N, C),
       T1 = scalarProductCoeffs(N, T)
```

Numeri razionali L'implementazione della soluzione che possa accettare i numeri razionali prevede l'introduzione del predicato rat/2 che possa rappresentare per ognu numero il proprio numeratore e denominatore. Per poter accomodare operazioni come somma, prodotto, sottrazione divisione c'è sarà bisogno di implementare le funzioni private per il Minimo comune multiplo e il massimo comune divisore.

isRat/2 è una funzione che permette di riconoscere un numero razionale ed essere sicuro che non rappresenti un differente struttura dati

```
isRat(rat(_, _))
```

Il minimo comune multiplo tra 2 numeri a e b rappresenta il più piccolo numero naturale intero positivo multiplo di entrambi:

$$\text{mcm}(a, b) = \frac{a \cdot b}{\text{MCD}(a, b)}$$

Il massimo comune divisore tra 2 numeri a e b rappresenta il più il numero naturale più grande per il quale entrambi possono essere divisi esattamente:

$$\text{MCD}(a, b) = \begin{cases} b, & \text{se } a \bmod b = 0, \\ \text{MCD}(b, a \bmod b), & \text{altrimenti.} \end{cases}$$

Nella mia implementazione in fabula ho deciso di implementare similmente le 2 funzioni.

```
1  ▷mcd/2 Massimo comun divisore
2  mcd(A, 0) = A
3  mcd(A, B) = mcd(B, A - B) if A > B
4  mcd(A, B) = mcd(A, B - A) if B > A
5  mcd(A, A) = A
6
7  mcm/2 Minimo comun multiplo
8  mcm(A, B) = (A*B) if mcd(A, B) = 1
9  mcm(A, B) = (A * B) / mcd(A, B)
```

Le seguenti sono funzioni che vanno a sovrascrivere il comportamento di default degli operatori per l'addizione ,prodotto e divisione presente in Fabula. Dato che si deve comunque poter effettuare operazioni con i numeri interi bisogna creare i vari casi utilizzando *integer* e *isRat* per controllare la natura dell'input

```
1  ▷ Operatore somma
2  '+'(X, Y) = rat(NR, DR)
3      if isRat(X), isRat(Y),
4      X = rat(NX, DX),
5      Y = rat(NY, DY),
6      DR = mcm(DX, DY),
7      NR = NX * (DR div DX) + NY * (DR div DY)
8
9  ▷ Operatore divisione
10 '/'(X, Y) = rat(X,Y)
11     if integer(X), integer(Y)
12 ▷ Operatore divisione
13 '/'(X, Y) = X * (1 / Y)
14     if isRat(X), isRat(Y)
15 ▷ Operatore divisione
16 '/'(Number, rat(N, D)) = rat(Number * D, N)
17     if integer(Number), isRat(rat(N, D))
18 ▷ Operatore divisione
19 '/'(rat(N, D), Number) = rat(N, D * Number)
20
21 ▷ Operatore prodotto
22 '*'(X, Y) = rat(NR, DR)
23     if isRat(X), isRat(Y),
24     X = rat(NX, DX),
25     Y = rat(NY, DY),
26     NR = NX * NY,
27     DR = DX * DY
28
29 ▷ Operatore prodotto
30 '*'(Number, rat(N, D)) = rat(Number * N, D)
31     if integer(Number), isRat(rat(N, D))
```

Se si prevede la possibilità di effettuare operazioni su numeri razionali soltanto con altri numeri razionali allora è sufficiente una implementazione degli operatori somma, prodotto e divisione. In caso in cui questa cosa non accada o si utilizzano diverse implementazioni a seconda della della posizione del numero oppure si passa tramite una funzione che converte i numeri nella rappresentazione razionale e nel caso lo sia già lo restituisce senza modificarlo. In entrambi i casi sarà comunque necessario avere una implementazione dell'operatore '/' che possa permettere di passare dalla rappresentazione numerica alla rappresentazione dei numeri razionali

```
1 ▷ toRat/2
2 toRat(Number) = rat(Number, 1) if integer(Number)
3 toRat(rat(N, D)) = rat(N, D)
```

Non sarà necessario avere una implementazione dell'operatore sotrazione dato che sfrutterà quella built in del linguaggio e la nuova operazione somma pensata per i razionali.

Esempi. Descrizione di alcuni esempi di utilizzo.

Densa a una variabile

1. **Esempio 1: Somma di due polinomi costanti**
   - `poly(x,0,[10]) + poly(x,0,[10])` come risultato si ottiene `poly(x,0,[20])`
2. **Esempio 2: Prodotto di due polinomi costanti**
   - `poly(x,0,[10]) * poly(x,0,[10])` come risultato si ottiene `poly(x,0,[100])`
3. **Esempio 3: Prodotto di un polinomio lineare con un polinomio costante**
   - `poly(x,1,[10, 10]) * poly(x,0,[10])` come risultato si ottiene `poly(x,1,[100,100])`
4. **Esempio 4: Somma di due polinomi di grado 4**
   - `poly(x,3,[0,0,0,1]) + poly(x,3,[0,0,0,1])` come risultato si ottiene `poly(x,3,[0,0,0,2])`
5. **Esempio 5: Somma di due polinomi di grado 4 con termini diversi**
   - `poly(x,4,[1,0,0,1]) + poly(x,4,[9,0,0,1])` come risultato si ottiene `poly(x,3,[10,0,0,2])`
6. **Esempio 6: Prodotto di due polinomi di grado 4**
   - `poly(x,3,[1,2,3,4]) * poly(x,3,[1,2,3,4])` come risultato si ottiene `poly(x,6,[1,4,10,20,25,24,16])`
7. **Esempio 7: Derivata di un polinomio di grado 4**
   - `diff(poly(x,3,[1,2,3,4]))` come risultato si ottiene `poly(x,2,[2,6,12])`

Sparsa a una variabile

1. **Esempio 1: Somma di un polinomio con un valore costante**
   - `x + 32` come risultato si ottiene `poly(x,0,[k(0,32)])` ->
2. **Esempio 2: Somma di due polinomi**
   - `poly(x,0,[k(0,32)]) + poly(x,0,[k(0,32)])` come risultato si ottiene `poly(x,0,[k(0,64)])`
3. **Esempio 3: Prodotto di due polinomi**
   - `poly(x,1,[k(0,2), k(1,2)]) * poly(x,1,[k(0,2), k(1,2)])` come risultato si ottiene `poly(x,2,[k(0,4),k(1,8),k(2,4)])`
4. **Esempio 4: Derivata di un polinomio**
   - `diff(poly(x,2,[k(0,1), k(1,2), k(2,4)]))` come risultato si ottiene `poly(x,1,[k(0,2),k(1,8)])`

Sparsa a più di una variabile

1. **Esempio 1: Somma tra 2 polinomi**
   - `poly(x, [k(1, 4), k(0, 3)]) + poly(y, [k(2, 5), k(0, 2)])` come risultato si ottiene `poly(x,[k(0,poly(y,[k(2,5),k(0,2)])),k(1,4),k(0,3)])`
   - `poly(x,[k(2,3),k(1,2),k(0,1)]) + poly(y,[k(3,4),k(1,1)])` come risultato si ottiene
     `poly(x,[k(0,poly(y,[k(3,4),k(1,1)])),k(2,3),k(1,2),k(0,1)])` $\rightarrow 4y^3 + 1y + 3x^2 + 2x + 1$
2. **Esempio 2: Prodotto tra 2 polinomi**
   - `poly(x, [k(0, 1), k(1, 1)]) * poly(x, [k(0, 1), k(1, 1)])` si ottiene *poly(x,[k(0,1),k(1,2),k(2,1)])*
   - `poly(y, [k(0, 1), k(1, 1)]) * poly(x, [k(0, 1), k(1, 1)])` come risultato si ottiene
     `poly(x,[k(0,poly(y,[k(0,1),k(1,1)])),k(1,poly(y,[k(0,1),k(1,1)]))])`
     $\rightarrow 1 + 1y + 1x + 1xy$
   - `poly(x, [k(0, 1), k(3, 2)]) * poly(x, [k(2, 3), k(5, 1)])` come risultato ottiene `poly(x,[k(2,3),k(5,7),k(8,2)])` $\rightarrow 3x^2 + 7x^5 + 2x^8$
3. **Esempio 3: Derivata di primo grado**
   - `diff(x, poly(x, [k(0, 2), k(1, 3), k(2, 4)]))` ha come risultato `poly(x,[k(0,3),k`
     $\rightarrow 3 + 8x$
   - `diff(y, poly(x, [k(0, 2), k(1, 3), k(2, 4)]))` $\rightarrow 0$
   - `diff(x, poly(x, [k(0, 1),k(1,2),k(2, poly(y, [k(0, 2),k(1, 1)]))]))` ha come risultato
     `poly(x,[k(0,2),k(1,poly(y,[k(0,4),k(1,2)]))])` $\rightarrow 2 + 4x + 2yx$

Numeri razionali

- `rat(1,2) + rat(1,3)` = rat(5,6)
- `rat(1,2) * rat(1,3)` = rat(1,6)
- `rat(1,2) / rat(1,3)` = rat(3,2)
- (toRat(2)) = rat(2,1)

Esercizio 5

Problema. L'ultimo problema consiste nell'implementazione di un interprete di un linguaggio logico simile al Prolog. Un interprete, in informatica e nella programmazione, è un programma in grado di eseguire altri programmi a partire direttamente dal relativo codice sorgente scritto in un linguaggio di alto livello.

Metodo. Prolog è un linguaggio di programmazione logica basato sul paradigma dichiarativo: invece di indicare come ottenere una soluzione, si definisce cosa è vero (fatti) e quando qualcosa può essere considerato vero (regole). Prolog si occupa di unificare una query con fatti e regole presenti nel programma.

Il prolog si basa sulle clausole d Horn:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B$$

Per l'implementazione di un interprete di un linguaggio funzionale ci sarà bisogno di leggere passo passo la testa della lista all'interno della query e verificare che essa sia vera tramite fatti e regole. Bisognerà prestare attenzione sull'implementazione del cut perché non esiste come funzionalità built-in del linguaggio.

Strutture dati. Le 2 strutture dati principali saranno clause/1 per rappresentare fatti e regole del linguaggio Prolog e goal/1 per rappresentare l'obiettivo che ci si è posti di validare.

- **Fatti**, costituiti da una testa (ad esempio `arc(1,2)`) il cui corpo è vuoto, poiché non richiede ulteriori prove.
- **Regole**, caratterizzate da una testa (ad esempio `path(X,Y, [X,Y])`) e da un corpo non vuoto (es. `[arc(X,Y)]`). Per la verifica della testa, è necessario dimostrare tutte le affermazioni nel corpo.
- **Goal (Interrogazioni)**, che specificano una lista di condizioni da provare (es. `goal([member(2, [1,2,3])])` o `goal([append([1,2], [3,4], R)])`).
- **Cut**, un operatore che, quando presente nel corpo di una regola, limita il backtracking, bloccando la ricerca di soluzioni alternative.

All'interno del programma ci sono 2 modi per poter rappresentare le clausole:

1. `clause(arc(1,2), [])`
2. `clausola(arc(X,Y), [[arc(1,2),[]]]`

Dato che Fabula è un linguaggio non deterministico la seconda versione viene sfruttata quando sopraggiunge la necessità di dare un ordine alla valutazione di fatti e regole

Funzioni.

Interprete senza il cut La funzione di base per poter implementare l'interprete del linguaggio simile al Prolog e permette di sfruttare le logiche dietro al linguaggio logico funzionale di Fabula per poter interpretare passo passo le clausole ed utilizzare una ricerca in profondità. La funzione goal è pubblica ed esposta all'utente finale.

```
▷ goal/1 funzione pubblica

goal([])
goal([true])
goal([Head | Rest]) = goal(Rest)
  if clause(Head, Tail), goal(Tail)
```

Nel caso in cui la lista dentro al goal sia vuota viene restituito *true*. In caso contrario si divide il problema in diverse teste da testare con le clausole e si prosegue ricorsivamente.

1. Si prende il primo obiettivo $Head$ dalla lista $[Head \mid Rest]$.
2. Si cerca una clausola nella forma `clause(`$Head$`, `$Tail$`)`:
   - Se esiste, si verifica il corpo $Tail$ chiamando ricorsivamente `goal(`$Tail$`)`.
   - Se `goal(`$Tail$`)` restituisce vero, si procede con la verifica della coda $Rest$ tramite `goal(`$Rest$`)`.
3. Il processo continua fino a quando tutti gli obiettivi nella lista sono stati verificati o finché una delle condizioni fallisce.

In questo modo si riesce ad unificare tutte le teste e le regole.

Alcune implementazioni da Prolog a clausole Fabula: Predicato path/3

```
path(X, Y, [X, Y]) :- arc(X, Y).
path(X, Y, [X | R]) :- arc(X, Z), path(Z, Y, R).

clause(path(X,Y, [X, Y]), [arc(X,Y)])
clause(path(X,Y, [X | R]), [arc(X,Z), path(Z,Y,R)])
```

Predicato member/2

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

clause(member(X, [X | _]), [])
clause(member(X, [_ | T]), [member(X, T)])
```

Predicato append/3

```
append([], R, R).
append([H|T], L, [H|R]) :- append(T, L, R).

clause(append([], R, R), [])
clause(append([H | T], L, [H | R]), [append(T, L, R)])
```

La sintassi nei 2 linguaggi si assomiglia molto, la semantica tranne per il cut è la medesima. Le clausole permettono di esprimere anche le regole omettendo di inserire elementi nella seconda lista.

```
clause(arc(1,2), [])
clause(arc(1,3), [])
clause(arc(2,4), [])
clause(arc(3,4), [])
clause(arc(4,5), [])
clause(arc(5,6), [])
```

In questo caso vengono definiti gli archi di un possibile grafo. Per poter sfruttare il *cut* c'è bisogno di implementare un sistema per poter avere un ordinamento nella valutazione delle clausole, essendo fabula un linguaggio logico funzionale non deterministico bisogna implementare

Interprete con il cut Nella implementazione che prevede il *cut* si passa alla seconda rappresentazione delle clausole dove vengono inserite all'interno di una lista comune per ogni categoria di fatti e regole.

```
clausola(arc(X,Y),
    [[arc(1,2),[]],
    [arc(1,3),[]],
    [arc(2,4),[]],
    [arc(3,4),[]]
    ])

clausola(path(X,Y,[X|R]),
    [[path(X,Y,[X,Y]), [arc(X,Y)]],
    [path(X,Y,[X|R]), [arc(X,Z),
    path(Z,Y,R)]] ])


clausola(cut, [[cut, []]])
```

L'unica funzione pubblica rimane il `goal/1` con cui l'utente dovrà sottoporre la propria query. Fabula non permette di fermare l'esenzione del programma per poter utilizzare correttamente il *cut* e fermare il backtracking, quello che si può implementare in questo caso è una risposta per l'utente che possa comunicare il fatto che sia arrivato al *cut* e che quindi da li in poi l'esecuzione del programma sia terminata.

```
▷ goal/1 funzione pubblica

goal([])
goal([true])
goal([H | R]) = goal(R)
  if Corrispondenza = trova_corrispondenza([H, _]),
     Corrispondenza = [H, B],
     H ≠ cut,
     goal(B)

print([H | R]) = trova_corrispondenza([H, _])

```

```
13 goal([cut | R]) = goal(R)
14   if Corrispondenza = trova_corrispondenza([cut, _])
15
16 goal([cut | _]) = stop
```

A livello verticale il back tracking può essere controllato tramite le nuove clausole che forzano lo scorrimento in una sola direzione.

```
1 trova_corrispondenza([Testa, Corpo]) = Corrispondenza
2   if clausola(Testa, Lista),
3   Corrispondenza = member1(Lista, [Testa, Corpo])
```

Nella versione base la funzione ottiene tutte le clausole corrispondenti ad una specifico caso e poi le restituisce una ad una.

Esempi. Esempi di utilizzo dell'interprete Prolog.

Il primo esempio vede l'utilizzo del predicato arc per poter scegliere un arco che va dal nodo 1 a qualsiasi altro nodo a lui collegato, il secondo predicato vede un percorso che va dal quest ultimo al nodo 4 seguendo un percorso descritto tramite la variabile P.
Risultati:

$$\text{goal}([\text{arc}(1, X), \text{path}(X, 4, P)]$$

$$X = 2, \qquad P = [2, 4]$$
$$X = 3, \qquad P = [3, 4]$$

Il secondo esempio è analogo al primo, ma si è preso 6 al posto di 4 permettendo così di avere un percorso tra 4 diversi nodi.
Risultati:

$$\text{goal}([arc(1, X), path(X, 4, P)])$$
$$X = 2, \quad P = [2, 4, 5, 6]$$

Il terzo uso il predicato member per elencare tutti gli elementi facenti parte della seconda lista
Risultati:

$$\text{goal}([member(X, [a, b, c])])$$
$$X = a, b, c$$

Il quarto esempio inserisco un cut all'interno del goal tra il primo e il secondo arco. Verrà restituito X = 2 e poi verrà stampato stop.
Risultati:

$$\text{goal}([arc(1, X), cut, arc(X, 4)])$$
$$X = 2, stop, ...$$

Il quinto esempio inserisco un cut all'interno del goal tra il primo arco e il path, quindi si potrà scegliere soltanto 1 percorso
Risultati:

$$\text{goal}([arc(1, X), cut, path(X, 4, P)])$$
$$P = [2, 4], stop, ...$$

Appendice

Alcune definizioni utilizzate all'interno della relazione:

Forward Chaining Iniziando dai dati disponibili e usa le regole di inferenza per ricavare ulteriori dati fino al raggiungimento di un certo obiettivo. Un motore inferenziale che usa Forward Chaining ad ogni passo va alla ricerca delle regole di inferenza tali per cui la premessa è nota essere vera, dopodiché può dedurre la conseguenza e aggiungerla come nuovo dato a disposizione.

Background Chaining Iniziando da una lista di obiettivi (o ipotesi) e ragiona in modo inverso rispetto alla Forward Chaining, partendo dalle conseguenze delle regole di inferenza disponibili e verificando se le premesse sono note essere vere.

Funzioni frequentemente utilizzate all'interno del documento

Le seguenti funzioni di supporto eseguono operazioni ricorrenti nella risoluzione del problema e sono state implementate per facilitare il loro riutilizzo in futuro.

member/2 Funzione per controllare se un elemento è presente all'interno di una lista. Nell'ultimo esercizio viene usata anche come mebmer1.

```
1 ▷ member/2 funzione privata
2 member([H | _], H) = true
3 member([_ | T], H) = member(T, H)
```

anyMember/2 Funzione per controllare se almeno un elemento del primo membro è presente nel secondo, sfrutta la funzione member/2.

```
1 ▷ anyMember/2 funzione privata
2 anyMember([H | _], L) = true if member(L, H)
3 anyMember([_ | T], L) = anyMember(T, L)
```

subset/2 Funzione per constatare se una lista è completamente contenuta in un'altra, utile quando bisogna confrontare 2 insiemi.

```
1 ▷ subset/2 funzione privata
2 subset([], _) = true
3 subset([H | T], S) if member(S, H), subset(T, S)
```

andList/2 Funzione per trasformare uno o più `and` annidati in una lista.

```
1 ▷ andList/2 funzione privata
2 andList(P) = andList1(P, [])
3 andList1(A, L)          = [A | L] if symbol(A)
4 andList1(and(A, B), L) = andList1(A, andList1(B, L))
```

orList/2 Funzione per trasformare uno o or `or` annidati in una lista.

```
1 ▷ orList/2 funzione privata
2 orList(P) = orList1(P, [])
3 orList1(A, L)          = [A | L] if symbol(A)
4 orList1(or(A, B), L) = orList1(A, orList1(B, L))
```

normalizeNot/1 Funzione che ha il compito di gestire il caso in cui una proposizione sia doppiamente negata.

```
1 ▷ normalizeNot/1 funzione privata
2 normalizeNot(not(not(P))) = P
3 normalizeNot(not(P)) = not(P)
4 normalizeNot(P) = P
```

removeElement/1 Funzione per rimuovere un elemento da una lista.

```
1 ▷ removeElement/1 funzione privata
2 removeElement([H | T], [H]) = T
3 removeElement([H | T], [X]) = [H | removeElement(T, [X])]
```

removeList/1 Funzione per rimuovere una sottolista da un'altra lista.

```
1 ▷ removeList/2 funzione privata
2 removeList(Stato, []) = Stato
3 removeList(Stato, [H | T]) =
4         removeList(removeElement(Stato, [H]), T)
```

checkNoDuplicates/1 Funzione per controllare la presenza di un elemento duplicato all'interno di un piano.

```
1 ▷ checkNoDuplicates/2 funzione privata
2 checkNoDuplicates(PlanParziale, Plan) = true
3     if not_member(PlanParziale, Plan)
```

not_member/2 Funzione per controllare iterativamente che ogni elemento della prima lista non sia presente nella seconda.

```
1 ▷ not_member/2 funzione privata
2 not_member([], _) = true
3 not_member([H | T], Plan) = not_member(T, Plan)
4     if not_in(H, Plan)
```

not_in/1 Funzione di appoggio per controllare che un elemento non faccia parte di una lista.

```
▷ not_in/2 funzione privata
not_in(_, []) = true
not_in(X, [H | T]) = not_in(X, T)
    if X ≠ H
```

remove_period/1 Funzione che prende in input una lista contenente dei termini ed elimina il punto finale se presente.

```
▷ remove_period/2 funzione privata
remove_period(['.']) = []
remove_period([X | Rest]) = [X | remove_period(Rest)]
```

Nei vari esercizi nel caso in cui venissero utilizzate funzioni private con la medesima firma non verranno riproposte una seconda volta e verrà utilizzato il precedente elenco come riferimento per la loro implementazione

Poly To Latex La seguente funzione privata permette di convertire i polinomi multiple variabili in un formato simile a quello utilizzato da latex, per facilitarne l'utilizzo in altri contesti.

```

toLatex(poly(Var, [])) = "0"
toLatex(poly(Var, [Term])) = toLatexTerm(Term, Var)
toLatex(poly(Var, [Term | Rest])) =
    toLatexTerm(Term, Var) + " + " + toLatex(poly(Var, Rest))

toLatexTerm(k(E, C), Var) = S
    if isPoly(C),
    S = toLatexDistribuito(E, Var, polyVar(C), polyCoeffs(C))

toLatexTerm(k(0, C), _) = toLatexCoeff(C)
toLatexTerm(k(1, C), Var) =
    toLatexCoeff(C) + varToString(Var)

toLatexTerm(k(E2, C), Var) =
    toLatexCoeff(C)
    + varToString(Var)
    + "^{" + stringOfInt(E2) + "}"

toLatexCoeff(C) = stringOfInt(C)
toLatexCoeff(poly(Var, CoeffList)) =
    toLatex(poly(Var, CoeffList))


toLatexDistribuito(_, _, _, []) = "0"
toLatexDistribuito(E, Var, SubVar, [k(SubE, SubC)]) =
```

```
27      subTerm(E, Var, SubE, SubC, SubVar)
28
29  toLatexDistribuito(E, Var, SubVar, [k(SubE, SubC) | Rest]) =
30      subTerm(E, Var, SubE, SubC, SubVar)
31      + " + "
32      + toLatexDistribuito(E, Var, SubVar, Rest)
33
34  subTerm(E, Var, SubE, SubC, SubVar) =
35      stringOfInt(SubC)
36      + varExponent(Var, E)
37      + varExponent(SubVar, SubE)
38
39  varExponent(_, 0) = ""
40  varExponent(Var, 1) = varToString(Var)
41  varExponent(Var, E) =
42      varToString(Var) + "^{" + stringOfInt(E) + "}"
43
44  var(x)
45  var(y)
46  var(z)
47
48  varToString(x) = "x"
49  varToString(y) = "y"
50  varToString(z) = "z"
51
52  stringOfInt(0) = "0"
53  stringOfInt(1) = "1"
54  stringOfInt(2) = "2"
55  stringOfInt(3) = "3"
56  stringOfInt(4) = "4"
57  stringOfInt(5) = "5"
58  stringOfInt(6) = "6"
59  stringOfInt(7) = "7"
60  stringOfInt(8) = "8"
61  stringOfInt(9) = "9"
62
63  isPoly(poly(_, _))
64
65  polyVar(poly(Var, _)) = Var
66  polyCoeffs(poly(_, Coeffs)) = Coeffs
```

L'obiettivo è stampare correttamente espressioni come

$$\text{poly}(x, \big[k(1, \text{poly}(y, [k(3,4), k(1,1)])),\ k(2,3)\big])$$

in modo che diventino

$$4\,xy^3 + 1\,xy + 3\,x^2$$

utilizzando la sintassi LaTeX.

La funzione `toLatex` riceve un polinomio e produce una stringa in stile LaTeX. Se la lista dei termini `CoeffList` è vuota, restituisce la stringa `"0"`. Se c'è più di un termine, concatenerà le conversioni di ciascun termine con `" + "`.

Questa funzione gestisce un singolo termine `k(E, C)`:

- Se `C` è un intero, utilizza `termWithExponent(E, C, Var)` (o equivalenti) per formare, ad esempio, `3x^2`.
- Se `C` è un polinomio annidato, chiama una funzione dedicata (`toLatexDistribuito`) che "distribuisce" correttamente `Var^E` dentro il polinomio annidato.

Dato un esponente `E`, una variabile esterna `Var` e un polinomio interno con variabile `SubVar`, produce una stringa risultante dall'espansione

$$\texttt{Var\^{}E} \times \big(\texttt{SubVar\^{}e1} \cdot c1 \ + \ \texttt{SubVar\^{}e2} \cdot c2 + \dots\big).$$

Viene gestita a livello di singoli termini, concatenandoli infine con `" + "`.

Costruisce la stringa di un singolo "pezzo" risultato dall'espansione. Per esempio, se `E=1`, `Var=x`, `SubE=3`, `SubC=4`, `SubVar=y`, il risultato sarà

$$4\,x\,y^3$$

Tale funzione può essere sicuramente migliorata, in questo caso sono andato a coprire soltanto i casi base e non sono andato ad intervenire sulla rappresentazione con i numeri razionali.