EBook Gratuito

APPRENDIMENTO
Prolog Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#prolog

Sommario

Di

Capitolo 1: Iniziare con Prolog Language

Osservazioni

implementazioni

1. SWI-Prolog (gratuito) swi-prolog
    - Implementato in c
2. SICStus (commerciale) sicstus-prolog
3. YAP (gratuito) yap
4. GNU Prolog (gratuito) gnu-prolog
5. XSB (gratuito) xsb
6. B (commerciale) b-prolog
7. IF (commerciale)
8. Ciao (gratuito)
9. Minerva (commerciale)
10. ECLiPSe-CLP (gratuito) eclipse-clp
11. Jekejeke Prolog (commerciale)
12. Prolog IV
13. Resa Prolog (gratuito)
    - Implementato in c # , javascript e phyton
14. Visual Prolog (commerciale) visual-prolog

Examples

Installazione o configurazione

SWI-Prolog

Windows e Mac:

- Scarica SWI-Prolog dal sito ufficiale
- Basta installare seguendo le istruzioni di installazione.

Linux (PPA):

- Aggiungi il `ppa:swi-prolog/stable` alle sorgenti software del tuo sistema (gli sviluppatori possono scegliere per `ppa:swi-prolog/devel`):
    - Aprire un terminale (Ctrl + Alt + T) e digitare: `sudo add-apt-repository ppa:swi-prolog/stable`
    - Successivamente, aggiorna le informazioni sul pacchetto: `sudo apt-get update`
- Ora installa SWI-Prolog tramite il gestore pacchetti: `sudo apt-get install swi-prolog`

- È ora possibile avviare SWI-Prolog tramite la riga di comando con il comando `swipl`

aggiungere / 3

```
append([], Bs, Bs).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).
```

`append/3` è una delle relazioni Prolog più conosciute. Definisce una relazione tra tre argomenti ed è vera *se* il terzo argomento è una lista che denota la concatenazione delle liste che sono specificate nel primo e nel secondo argomento.

In particolare, e come è tipico per il buon codice Prolog, `append/3` può essere utilizzato in *più direzioni* : Può essere usato per:

- *aggiungere* due elenchi completamente o parzialmente istanziati:

  ```
  ?- A = [1, 2, 3], B=[4, 5, 6], append(A, B, Y)
  Output:
  A = [1, 2, 3],
  B = [4, 5, 6],
  Y = [1, 2, 3, 4, 5, 6].
  ```

- *controlla* se la relazione è vera per tre elenchi completamente istanziati:

  ```
  ?- A = [1, 2, 3], B = [4, 5], C = [1, 2, 3, 4, 5, 6], append(A, B, C)
  Output:
  false
  ```

- *generare* tutti i modi possibili per aggiungere due elenchi a una determinata lista:

  ```
  ?- append(A, B, [1, 2, 3, 4]).
  Output:
  A = [],
  B = [1, 2, 3, 4] ;
  A = [1],
  B = [2, 3, 4] ;
  A = [1, 2],
  B = [3, 4] ;
  A = [1, 2, 3],
  B = [4] ;
  A = [1, 2, 3, 4],
  B = [] ;
  false.
  ```

Vincoli CLP (FD)

I vincoli CLP (FD) sono forniti da tutte le implementazioni seria di Prolog. Ci permettono di ragionare sugli **interi** in modo puro.

```
?- X #= 1 + 2.
X = 3.
```

```
?- 5 #= Y + 2.
Y = 3.
```

Programmazione del database

Prolog classifica tutto in:

- **Atomi** - Qualsiasi sequenza di caratteri che non iniziano con un alfabeto maiuscolo. Ad esempio - `a` , `b` , `okay`
- **Numeri** - Non esiste una sintassi speciale per i numeri, non è richiesta alcuna dichiarazione. Ad esempio `1` , `22` , `35.8`
- **Variabili** - Una stringa che inizia con un carattere maiuscolo o underscore (_). Ad esempio `X` , `Y` , `Abc` , `AA`
- **Termini complessi** : sono costituiti da un *funtore* e da una sequenza di *argomenti* . Il nome di un termine complesso è sempre un atomo, mentre gli argomenti possono essere atomi o variabili. Ad es. `father(john,doe)` , `relative(a)` , `mother(X,Y)` .

Un database logico contiene un insieme di *fatti* e *regole* .

Un termine complesso con solo gli atomi come argomenti è chiamato un fatto, mentre un termine complesso con variabili come argomenti è chiamato una regola.

Esempio di fatti in Prolog:

```
father_child(fred, susan).
mother_child(hillary, joe).
```

Esempio di una regola in Prolog:

```
child_of(X,Y):-
    father_child(Y,X)
    ;
    mother_child(Y,X).
```

Si noti che il `;` qui è come l'operatore `or` in altre lingue.

Prolog è un linguaggio dichiarativo e puoi leggere questo database come segue:

> *fred è il padre di susan*
>
> *hillary è la madre di joe.*
>
> *Per tutti `X` e `Y` , `X` è figlio di `Y` se `Y` è padre di `X` o `Y` è madre di `X`*

In effetti, un insieme finito di fatti e / o regole costituisce un *programma* logico.

L'uso di tale programma è dimostrato facendo *domande* . Le query ti consentono di recuperare informazioni da un programma di logica.

Per caricare il database nell'interprete (supponendo di aver salvato il database nella directory in cui stai eseguendo l'interprete), inserisci semplicemente:

```
?- [nameofdatabase].
```

sostituendo `nameofdatabase` con il nome file effettivo (si noti che qui si esclude l'estensione `.pl` al nome file).

Esempio di query nell'interprete per il programma sopra e i risultati:

```
?- child_of(susan,fred).
true

?- child_of(joe,hillary).
true

?- child_of(fred,susan).
false

?- child_of(susan,hillary).
false

?- child_of(susan,X).
X = fred

?- child_of(X,Y).
X = susan,
Y = fred ;
X = joe,
Y = hillary.
```

Le domande sopra e le loro risposte possono essere lette come segue:

> *è Susan un figlio di Fred? - vero*
>
> *è un figlio di Hillary? - vero*
>
> *Fred è figlio di Susan? - falso*
>
> *è Susan un figlio di Hillary? - falso*
>
> *di chi è Susan? - Fred*

Ecco come programmiamo la logica in Prolog. Un programma logico è più formalmente: un insieme di assiomi, o regole, che definiscono le relazioni (noti come predicati) tra gli oggetti. Un modo alternativo di interpretare il database in un modo logico più formale è:

> *La relazione `father_child` tiene tra fred e susan*
>
> *La relazione `mother_child` tiene tra hillary e joe*
>
> *Per tutte le `X` e `Y` la relazione `child_of` tiene tra `X` e `Y` se la relazione `father_child` contiene tra `Y` e `X` , o la relazione `mother_child` contiene tra `Y` e `X`*

Ciao, Mondo nell'interprete interattivo

Per stampare "Hello, World!" nell'interprete Prolog (qui stiamo usando `swipl` , la shell per SWI Prolog):

```
$ swipl
<...banner...>
?- write('Hello, World!'), nl.
```

`?-` è il prompt di sistema: indica che il sistema è pronto per l'utente per inserire una sequenza di *obiettivi* (cioè una *query*) che deve essere terminata con a `.` (punto).

Qui la query `write('Hello World!'), nl` ha due obiettivi:

- `write('Hello World!')` : `'Hello World!'` deve essere visualizzato **e** (`,`)
- una nuova linea (`nl`) deve seguire.

`write/1` (il `/1` è usato per indicare che il predicato accetta un argomento) e `nl/0` sono *predicati incorporati* (la definizione è fornita in anticipo dal sistema Prolog). I predicati incorporati forniscono strutture che non possono essere ottenute con la pura definizione Prolog o per evitare che il programmatore debba definirle.

L'output:

> Ciao mondo!
>
> sì

termina con `yes` il che significa che la query è riuscita. In alcuni sistemi è stampato `true` invece di `yes` .

Ciao, Mondo da un file

Apri un nuovo file chiamato `hello_world.pl` e inserisci il seguente testo:

```
:- initialization hello_world, halt.

hello_world :-
    write('Hello, World!'), nl.
```

La direttiva di `initialization` specifica che l'obiettivo `hello_world, halt` dovrebbe essere chiamato quando il file viene caricato. `halt` termina il programma.

Questo file può quindi essere eseguito dal tuo eseguibile Prolog. Le bandiere esatte dipendono dal sistema Prolog. Se stai usando SWI Prolog:

```
$ swipl -q -l hello_world.pl
```

Questo produrrà l'output `Hello, World!` . Il flag `-q` sopprime il banner che di solito viene visualizzato quando chiami esegui `swipl` . `-l` specifica un file da caricare.

Leggi Iniziare con Prolog Language online: https://riptutorial.com/it/prolog/topic/1038/iniziare-con-prolog-language

Capitolo 2: Alberi di derivazione

Examples

Albero di prova

L'albero delle prove (anche albero di ricerca o albero di derivazione) è un albero che mostra l'esecuzione di un programma Prolog. Questo albero aiuta a visualizzare il processo cronologico di backtracking presente in Prolog. La radice dell'albero rappresenta la query iniziale e i rami vengono creati quando si verificano punti di scelta. Ogni nodo dell'albero rappresenta quindi un obiettivo. I rami diventano foglie solo quando è stato dimostrato che sia vero / falso per l'obiettivo (set di) richiesto (s) e la ricerca in Prolog viene eseguita in modo **da profondità** a **sinistra da destra a prima** .

Considera il seguente esempio:

```
% Facts
father_child(paul,chris).        % Paul is the father of Chris and Ellen
father_child(paul,ellen).
mother_child(ellen,angie).       % Ellen is the mother of Angie and Peter
mother_child(ellen,peter).


% Rules
grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    father_child(Z,Y).

grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    mother_child(Z,Y).
```

Quando interrogiamo ora:

```
?- grandfather_grandchild(paul,peter).
```

la seguente struttura di prova visualizza il processo di ricerca in profondità:

```
                                  ?- grandfather_grandchild(paul,peter).
                                      /                                \
                                     /                                  \
  ?- father_child(paul,Z1),father_child(Z1,peter).            ?-
father_child(paul,Z2),mother_child(Z2,peter).
             /                    \                                    /
\
      {Z1=chris}              {Z1=ellen}                          {Z2=chris}
{Z2=ellen}
           /                        \                                /
\
?- father_child(chris,peter).  ?- father_child(ellen,peter).  ?- mother_child(chris,peter). ?-
mother_child(ellen,peter).
```

```
          |                          |                                |
/               \
       fail                       fail                            fail
fail(*)           success
```

(*) fallisce per `mother_child(ellen,angie)` dove 'angie' non riesce ad abbinare 'peter'

Leggi Alberi di derivazione online: https://riptutorial.com/it/prolog/topic/3097/alberi-di-derivazione

Capitolo 3: Elenchi di differenze

introduzione

Le liste delle differenze in Prolog denotano il concetto di conoscere la struttura di un elenco *fino a un certo punto* . Il resto dell'elenco può essere lasciato libero fino alla valutazione completa di un predicato. Una lista in cui la sua fine è sconosciuta viene indicata come una *lista aperta* , terminata da un *buco* . Questa tecnica è particolarmente utile per convalidare sintassi o grammatiche complesse.

La nota Definite Clause Grammars (DCG) utilizza le liste delle differenze per operare sotto il cofano.

Examples

Utilizzo di base

Consideriamo il predicato `sumDif/2` , verificato se la struttura di una lista corrisponde a diversi vincoli. Il primo termine rappresenta l'elenco da analizzare e il secondo termine un altro elenco che contiene la parte del primo elenco che è sconosciuta ai nostri vincoli.

Per la dimostrazione, `sumDif/2` riconosce un'espressione aritmetica per sommare *n* interi.

```
sumDif([X, +|OpenList], Hole) :-
    integer(X),
    sumDif(OpenList, Hole).
```

Sappiamo che il primo elemento della lista da validare è un numero intero, qui illustrato da `X` , seguito dal simbolo dell'aggiunta (`+`). Il resto dell'elenco che deve ancora essere elaborato in seguito (`OpenList`) non viene `OpenList` a quel livello. `Hole` rappresenta la parte dell'elenco che *non è necessario* convalidare.

Diamo un'altra definizione del predicato `sumDif/2` per completare la convalida dell'espressione aritmetica:

```
sumDif([X|Hole], Hole) :-
    integer(X).
```

Prevediamo un intero chiamato `X` direttamente all'inizio della lista aperta. È interessante notare che il resto dell'elenco `Hole` è rimasto sconosciuto e questo è l'intero scopo delle liste delle differenze: la struttura dell'elenco è nota fino a un certo punto.

Infine, il pezzo mancante arriva quando viene valutato un elenco:

```
?- sumDif([1,+,2,+,3], []).
true
```

Questo è quando si usa il predicato che la fine della lista è menzionata, qui `[]` , indica che la lista non contiene elementi aggiuntivi.

Valuta un'espressione aritmetica

Definiamo una grammatica che ci consenta di eseguire aggiunte, moltiplicazioni con l'uso di parentesi. Per aggiungere più valore a questo esempio, calcoleremo il risultato dell'espressione aritmetica. Riassunto della grammatica:

espressione → tempi
espressione → tempi '+' espressione
volte → elemento
volte → elemento '*' volte
elemento → "intero"
elemento → '(' espressione ')'

Tutti i predicati hanno un'arità di 3, perché hanno bisogno di aprire la lista, il buco e il valore dell'espressione aritmetica.

```
expression(Value, OpenList, FinalHole) :-
    times(Value, OpenList, FinalHole).

expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).

times(Value, OpenList, FinalHole) :-
    element(Value, OpenList, FinalHole).

times(TimesValue, OpenList, FinalHole) :-
    element(Value1, OpenList, ['*'|Hole1]),
    times(Value2, Hole1, FinalHole),
    TimesValue is Value1 * Value2.

element(Value, [Value|FinalHole], FinalHole) :-
    integer(Value).

element(Value, ['('|OpenList], FinalHole) :-
    expression(Value, OpenList, [')'|FinalHole]).
```

Per spiegare correttamente il principio dei *buchi* e come viene calcolato il valore, prendiamo la seconda `expression` frase:

```
expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).
```

L'elenco aperto è indicato dal predicato `OpenList` . Il primo elemento da convalidare è *ciò che precede il simbolo di addizione (`+`)* . Quando il primo elemento è convalidato, è seguito direttamente dal simbolo di addizione e dal seguito dell'elenco, chiamato `Hole1` . Sappiamo che

`Hole1` è l'elemento successivo da convalidare e può essere un'altra `expression` , quindi `Hole1` è quindi il termine assegnato `expression` del predicato.

Il valore è sempre rappresentato nel primo termine. In questa clausola, è definita dalla somma di `Value1` (tutto prima del simbolo di addizione) e `Value2` (tutto dopo il simbolo di addizione).

Infine, l'espressione può essere valutata.

```
?- expression(V, [1,+,3,*,'(',5,+,5,')'], []).
V = 31
```

Leggi Elenchi di differenze online: https://riptutorial.com/it/prolog/topic/9414/elenchi-di-differenze

Capitolo 4: Gestione degli errori ed eccezioni

Examples

Errori di istanziazione

Un errore di **istanziazione** viene generato se un argomento non è sufficientemente *istanziato* .

In modo critico, un errore di istanziazione *non può* essere sostituito da un *errore silenzioso* : *fallire* in questi casi significherebbe che non **c'è soluzione** , mentre un errore di istanziazione significa che *un'istanza* dell'argomento può partecipare a una soluzione.

Ciò è in contrasto con, ad esempio, l' **errore del dominio** , che può essere sostituito **dall'errore** silenzioso senza modificare il significato dichiarativo di un programma.

Punti generali sulla gestione degli errori

Prolog presenta **eccezioni** , che fanno parte dello standard ISO Prolog.

Un'eccezione può essere lanciata con `throw/1` e catturata con `catch/3` .

Lo standard ISO definisce molti casi in cui gli errori devono o possono essere lanciati. Le eccezioni standardizzate sono tutte l' `error(E,_)` modulo `error(E,_)` , dove `E` indica l'errore. Gli esempi sono `instantiation_error` , `domain_error` e `type_error` , che vedono.

Un predicato importante in relazione alle eccezioni è `setup_call_cleanup/3` , che vedete.

Pulizia dopo le eccezioni

Il predicato `setup_call_cleanup/3` , che viene attualmente considerato per l'inclusione nello standard ISO di Prolog e fornito da un numero crescente di implementazioni, ci consente di assicurare che le risorse siano correttamente liberate dopo che è stata generata un'eccezione.

Una tipica invocazione è:

```
setup_call_cleanup(open(File, Mode, Stream), process_file(File), close(Stream))
```

Si noti che un'eccezione o un'interruzione può verificarsi anche immediatamente dopo l' `open/3` in questo caso. Per questo motivo, la fase di `Setup` viene eseguita *atomicamente* . Nei sistemi Prolog che forniscono solo `call_cleanup/2` , questo è molto più difficile da esprimere.

Digitare e gli errori di dominio

Si verifica un **errore di tipo** se un argomento non è del *tipo* previsto. Esempi di tipi sono:

- `integer`
- `atom`

- `list` .

Se il predicato è del tipo previsto, ma al di fuori del *dominio* previsto, viene generato un **errore del dominio** .

Ad esempio, un errore di dominio è ammissibile se è previsto un numero intero compreso tra 0 e 15, ma l'argomento è l'intero 20.

Dichiaratamente, un errore di tipo o dominio equivale a un *errore silenzioso* , poiché nessuna istanziazione può creare un predicato il cui argomento è del tipo o del dominio errati.

Leggi Gestione degli errori ed eccezioni online: https://riptutorial.com/it/prolog/topic/7114/gestione-degli-errori-ed-eccezioni

Capitolo 5: Grammatiche di clausole definite (DCG)

Examples

Niente affatto: `... // 0`

Uno dei più elementari DCG non-terminale è `... //0` , che può essere letto come "qualcosa":

```
... --> [] | [_], ... .
```

Può essere usato per descrivere una lista `Ls` che contiene l'elemento `E` tramite:

```
phrase(( ..., [E], ... ), Ls)
```

Analisi con DCG

I DCG possono essere utilizzati per l'analisi. Meglio di tutti, *lo stesso* DCG può essere spesso usato per analizzare e *generare* liste che vengono descritte. Per esempio:

```
sentence --> article, subject, verb, object.

article --> [the].

subject --> [woman] | [man].

verb --> [likes] | [enjoys].

object --> [apples] | [oranges].
```

Query di esempio:

```
?- phrase(sentence, Ls).
Ls = [the, woman, likes, apples] ;
Ls = [the, woman, likes, oranges] ;
Ls = [the, woman, enjoys, apples] .

?- phrase(sentence, [the,man,likes,apples]).
true .
```

Obiettivi extra

Gli obiettivi extra consentono di aggiungere l'elaborazione alle clausole DCG, ad esempio, condizioni che gli elementi della lista devono soddisfare.

Gli obiettivi extra vengono osservati tra parentesi graffe alla fine di una clausola DCG.

```
% DCG clause requiring an integer
int --> [X], {integer(X)}.
```

Uso:

```
?- phrase(int, [3]).
true.

?- phrase(int, [a]).
false.
```

Argomenti aggiuntivi

Gli argomenti extra aggiungono risultati ai predicati di una clausola DCG, decorando l'albero di derivazione. Ad esempio, è possibile creare una grammatica algebrica che calcoli il valore alla fine.

Data una grammatica che supporta l'aggiunta dell'operazione:

```
% Extra arguments are passed between parenthesis after the name of the DCG clauses.
exp(C) --> int(A), [+], exp(B), {plus(A, B, C)}.
exp(X) --> int(X).
int(X) --> [X], {integer(X)}.
```

Il risultato di questa grammatica può essere convalidato e interrogato:

```
?- phrase(exp(X), [1,+,2,+,3]).
X = 6 ;
```

Leggi Grammatiche di clausole definite (DCG) online:
https://riptutorial.com/it/prolog/topic/2426/grammatiche-di-clausole-definite--dcg-

Capitolo 6: Linee guida per la codifica

Examples

Naming

Durante la programmazione in Prolog, dobbiamo scegliere due tipi di nomi:

- nomi di **predicati**
- nomi di **variabili** .

Un buon nome *predicato* chiarisce cosa significa ogni argomento. Per convenzione, i caratteri di **sottolineatura** vengono utilizzati nei nomi per separare la descrizione di diversi argomenti. Questo perché `underscores_keep_even_longer_names_readable` , mentre `mixingTheCasesDoesNotDoThisToTheSameExtent` .

Esempi di nomi di predicati validi sono:

- `parent_child/2`
- `person_likes/2`
- `route_to/2`

Si noti che vengono utilizzati nomi *descrittivi* . Gli imperativi sono evitati. L'uso di nomi descrittivi è consigliabile perché i predicati Prolog possono essere tipicamente utilizzati in *più* direzioni e il nome dovrebbe essere applicabile anche a tutti o nessuno degli argomenti è istanziato.

La capitalizzazione mista è più comune quando si selezionano i nomi delle *variabili* . Ad esempio: `BestSolutions` , `MinElement` , `GreatestDivisor` . Una convenzione comune per denominare le variabili che denotano *stati* successivi utilizza `S0` , `S1` , `S2` , ..., `S` , dove `S` rappresenta lo stato finale.

dentellatura

Ci sono solo pochi costrutti linguistici in Prolog e diversi modi per indentarli sono comuni.

Indipendentemente dallo stile scelto, un principio che deve essere sempre rispettato è di **non** posizionare **mai** `(;)/2` alla *fine* di una riga. Questo perché `;` e `,` aspetto molto simile, e `,` frequente `,` si verifica alla fine di una linea. Pertanto, le clausole che utilizzano una disgiunzione dovrebbero, ad esempio, essere scritte come:

```
(  Goal1
;  Goal2
)
```

Ordine degli argomenti

Idealmente, i predicati Prolog possono essere utilizzati in tutte le direzioni. Per molti predicati puri, anche questo è effettivamente il caso. Tuttavia, alcuni predicati funzionano solo in particolari

modalità , il che significa schemi di istanziazione dei loro argomenti.

Per convenzione, l'ordine di argomenti più comune per tali predicati è:

- **gli** argomenti di **input** vengono inseriti per primi. Questi argomenti devono essere istanziati *prima che* venga chiamato il predicato.
- *le coppie* di argomenti che appartengono insieme sono posizionate in modo adiacente, come `p(..., State0, State, ...)`
- gli argomenti di **output** previsti sono posti per ultimi. Questi predicati sono istanziati dal predicato.

Leggi Linee guida per la codifica online: https://riptutorial.com/it/prolog/topic/4612/linee-guida-per-la-codifica

Capitolo 7: Monotonicità

Examples

Ragionamento sui predicati monotonici

I predicati **monotonici** possono essere sottoposti a debug applicando il ragionamento *dichiarativo* .

In puro Prolog, un errore di programmazione può portare a uno o tutti i seguenti fenomeni:

1. il predicato non *riesce* correttamente in un caso in cui dovrebbe *fallire*
2. il predicato non *riesce* correttamente in un caso in cui dovrebbe *riuscire*
3. il predicato *loop* inaspettatamente dove dovrebbe produrre soltanto un insieme finito di risposte.

Ad esempio, considera come possiamo eseguire il debug di case (2) per ragionamento dichiarativo: possiamo sistematicamente *rimuovere gli* obiettivi delle clausole del predicato e vedere se la query non riesce *ancora* . Nel codice monotono, la rimozione degli obiettivi può al massimo rendere il programma risultante *più generale* . Quindi, possiamo individuare gli errori vedendo quale degli obiettivi porta al fallimento imprevisto.

Esempi di predicati monotoni

Esempi di predicati **monotoni** sono:

- **unificazione** con `(=)/2` o `unify_with_occurs_check/2`
- `dif/2` , che esprime disuguaglianza di termini
- **Vincoli CLP (FD)** come `(#=)/2` e `(#>)/2` , utilizzando una modalità di esecuzione monotona.

I predicati Prolog che usano solo obiettivi monotonici sono essi stessi monotoni.

I predicati monotonici consentono ragionamenti dichiarativi:

1. Aggiungere un vincolo (cioè un obiettivo) a una query può al massimo *ridurre* , mai estendere, l'insieme di soluzioni.
2. La rimozione di un obiettivo di tali predicati può al massimo *estendere* , mai ridurre, l'insieme di soluzioni.

Predicati non monotoni

Ecco alcuni esempi di predicati che **non** sono monotoni:

- predicati meta-logici come `var/1` , `integer/1` ecc.
- predicati di confronto termine come `(@<)/2` e `(@>=)/2`
- predicati che usano `!/0` , `(\+)/1` e altri costrutti che rompono la monotonicità
- *predicati di tutte le soluzioni* come `findall/3` e `setof/3` .

Se si utilizzano questi predicati, l' *aggiunta di* obiettivi può portare a più soluzioni, che va contro l'importante proprietà dichiarativa nota dalla logica che l'aggiunta di vincoli può al massimo *ridurre* , mai estendere, l'insieme di soluzioni.

Di conseguenza, anche altre proprietà che facciamo affidamento per il debugging dichiarativo e altri ragionamenti sono interrotte. Ad esempio, i predicati non monotonici infrangono la nozione fondamentale di **commutatività** della congiunzione nota dalla logica del primo ordine. Il seguente esempio illustra questo:

```
?- var(X), X = a.
X = a.

?- X = a, var(X).
false.
```

I predicati di tutte le soluzioni come `findall/3` interrompono anche la monotonicità: l' *aggiunta di* clausole può portare al *fallimento* degli obiettivi che in precedenza *erano stati mantenuti* . Ciò contrasta anche con la montonicità come nota dalla logica del primo ordine, dove l' *aggiunta di* fatti può al massimo *aumentare* , senza mai *ridurre* l'insieme di conseguenze.

Alternative monotone per costrutti non monotoni

Ecco alcuni esempi di come utilizzare predicati **monotonici** *invece* di costrutti impuri e non monotonici nei tuoi programmi:

- `dif/2` è pensato per essere usato al *posto* di costrutti non monotoni come `(\=)/2`
- i **vincoli** aritmetici (CLP (FD), CLP (Q) e altri) devono essere utilizzati al *posto* dei predicati aritmetici modificati
- `!/0` porta quasi sempre a programmi non monotoni e dovrebbe essere **evitato del** tutto.
- **errori di istanziazione** possono essere sollevati in situazioni in cui non è possibile prendere una decisione corretta in questo momento.

Combinando la monotonicità con l'efficienza

Talvolta si sostiene che, per ragioni di efficienza, dobbiamo accettare l'uso di costrutti non monotonici nei programmi Prolog del mondo reale.

Non ci sono prove per questo. Ricerche recenti indicano che il sottoinsieme monoprono puro di Prolog potrebbe non solo essere sufficiente per esprimere la maggior parte dei programmi del mondo reale, ma anche accettabilmente efficiente nella pratica. Un costrutto che è stato scoperto di recente e incoraggia questa visione è `if_/3` : Combina la monotonicità con una riduzione dei punti di scelta. Vedi **Indicizzazione dif / 2** .

Ad esempio, il codice del modulo:

```
pred(L, Ls) :-
    condition(L),
    then(Ls).
pred(L, Ls) :-
```

```
    \+ condition(L),
    else(Ls).
```

Può essere scritto con `if_/3` come:

```
pred(L, Ls) :-
    if_(condition(L),
        then(Ls),
        else(Ls)).
```

e *combina la* monotonicità con il determinismo.

Leggi Monotonicità online: https://riptutorial.com/it/prolog/topic/3989/monotonicita

Capitolo 8: operatori

Examples

Operatori predefiniti

Operatori predefiniti secondo ISO / IEC 13211-1 e 13211-2:

Priorità	genere	Operator (s)	Uso
1200	XFX	`:- -->`	
1200	fx	`:- ?-`	Direttiva, query
1100	xfy	`;`	
1050	xfy	`->`	
1000	xfy	`','`	
900	fy	`\+`	
700	XFX	`= \=`	Termine unificazione
700	XFX	`== \== @< @=< @> @>=`	Termine confronto
700	XFX	`=..`	
700	XFX	`is =:= =\= < > =< >=`	Valutazione e confronto aritmetico
600	xfy	`:`	Qualifica del modulo
500	YFX	`+ - /\ \/`	
400	YFX	`* / div mod // rem << >>`	
200	XFX	`**`	Potere galleggiante
200	xfy	`^`	Quantificazione variabile, potenza intera
200	fy	`+ - \`	Identità aritmetica, negazione; complemento bit per bit

Molti sistemi forniscono ulteriori operatori come un'estensione specifica dell'implementazione:

Priorità	genere	Operator (s)	Uso
1150	fx	`initialization multifile dynamic discontiguous`	Direttive standard
1150	fx	`mode block volatile public meta_predicate`	
900	fy	`spy nospy`	

Dichiarazione dell'operatore

In Prolog, gli operatori personalizzati possono essere definiti utilizzando `op/3` :

```
op(+Precedence, +Type, :Operator)
```

- Dichiara l'operatore come operatore di un tipo con una precedenza. L'operatore può anche essere un elenco di nomi nel qual caso tutti gli elementi dell'elenco sono dichiarati operatori identici.
- La precedenza è un numero intero compreso tra 0 e 1200, dove 0 rimuove la dichiarazione.
- Tipo è uno di: `xf` , `yf` , `xfx` , `xfy` , `yfx` , `fy` o `fx` dove `f` indica la posizione del funtore e `x` ed `y` indicano le posizioni degli argomenti. `y` denota un termine con una precedenza inferiore o uguale alla precedenza del funtore, mentre `x` denota una precedenza strettamente inferiore.
    - Prefisso: `fx` , `fy`
    - Infix: `xfx` (non associativo), `xfy` (associativo a destra), `yfx` (associativo a sinistra)
    - Postfix: `xf` , `yf`

Esempio di utilizzo:

```
:- op(900, xf, is_true).

X_0 is_true :-
  X_0.
```

Query di esempio:

```
?- dif(X, a) is_true.
dif(X, a).
```

Ordinamento a termine

Due termini possono essere confrontati tramite l'ordinamento standard:

> variabili @ <numeri @ <atomi @ <stringhe @ <strutture @ <liste

Gli appunti:

- Le strutture si confrontano alfabeticamente prima con il functor, poi con l'arity e infine con il confronto di ogni argomento.

- Gli elenchi confrontano prima per lunghezza, quindi per ciascun elemento.

Ordina l'operatore	Succede se
X @ <Y	X è inferiore a Y nell'ordine standard
X @> Y	X è maggiore di Y nell'ordine standard
X @ = <Y	X è minore o uguale a Y nell'ordine standard
X @> = Y	X è maggiore o uguale a Y nell'ordine standard

Query di esempio:

```
?- alpha @< beta.
true.

?- alpha(1) @< beta.
false.

?- alpha(X) @< alpha(1).
true.

?- alpha(X) @=< alpha(Y).
true.

?- alpha(X) @> alpha(Y).
false.

?- compound(z) @< compound(inner(a)).
true.
```

Termine uguaglianza

Operatore di uguaglianza	Succede se
X = Y	X può essere unificato con Y
X \ = Y	X non può essere unificato con Y
X == Y	X e Y sono identici (cioè si uniscono *senza* che si verifichino legami variabili)
X \ == Y	X e Y non sono identici
X =: = Y	X e Y sono aritmeticamente uguali
X = \ = Y	X e Y non sono aritmeticamente uguali

Leggi operatori online: https://riptutorial.com/it/prolog/topic/2479/operatori

Capitolo 9: Predicati extra logici

Examples

Predicati con effetti collaterali

I predicati che producono **effetti collaterali** lasciano il regno della pura logica. Questi sono ad esempio:

- `writeq/1`
- `read/1`
- `format/2`

Gli effetti collaterali sono fenomeni che non possono essere ragionati all'interno del programma. Ad esempio, eliminazione di un file o output sul terminale di sistema.

Predicati meta-logici

Predice che la ragione delle *istanziazioni* è chiamata **meta-logica** . Gli esempi sono:

- `var/1`
- `ground/1`
- `integer/1`

Questi predicati sono al di fuori del regno dei programmi di logica monotona pura, perché rompono proprietà come la *commutatività* della congiunzione.

Altri predicati che sono meta-logici includono:

- `arg/3`
- `functor/3`
- `(=..)/2`

Questi predicati potrebbero *in linea di principio* essere modellati nella logica del primo ordine, ma richiedono un numero infinito di clausole.

Prassi di tutte le soluzioni

Predice che la ragione di *tutte le soluzioni* sia extra-logica. Questi sono ad esempio:

- `setof/3`
- `findall/3`
- `bagof/3`

! / 0 e predicati correlati

I predicati che impediscono o proibiscono una lettura **dichiarativa** dei programmi Prolog sono extra-logici. Esempi di tali predicati sono:

- `!/0`
- `(->)/2` e se-allora-else
- `(\+)/1`

Questi predicati possono essere compresi solo proceduralmente, tenendo conto del reale flusso di controllo dell'interprete, e come tali sono al di là del regno della pura logica.

Leggi Predicati extra logici online: https://riptutorial.com/it/prolog/topic/2282/predicati-extra-logici

Capitolo 10: Prestazione

Examples

Macchina astratta

Per efficienza, il codice Prolog viene in genere compilato per **astrarre il codice macchina** prima che venga eseguito.

Sono state proposte molte diverse architetture e varianti di macchine astratte per l'esecuzione efficiente dei programmi Prolog. Questi includono:

- **WAM** , la *macchina astratta di Warren*
- **TOAM** , una macchina astratta usata in B-Prolog.
- **ZIP** , utilizzato ad esempio come base per la VM di SWI-Prolog
- **VAM** , un'architettura di ricerca sviluppata a Vienna.

indicizzazione

Tutti gli interpreti Prolog ampiamente utilizzati utilizzano l' **indicizzazione degli argomenti** per selezionare in modo efficiente le clausole appropriate.

Gli utenti possono tipicamente fare affidamento almeno *sull'argomento del primo argomento* , il che significa che le frasi possono essere efficacemente distinte dal funtore e dall'aritma del termine più esterno del *primo* argomento. Nelle chiamate in cui tale argomento è sufficientemente istanziato, le clausole di corrispondenza possono essere essenzialmente selezionate in tempo *costante* tramite hashing su quell'argomento.

Più recentemente, l' **indicizzazione JIT** è stata implementata in più sistemi, consentendo l'indicizzazione dinamica su qualsiasi argomento che sia sufficientemente istanziato quando viene chiamato il predicato.

Ottimizzazione della chiamata di coda

Praticamente tutti i sistemi Prolog implementano l' **ottimizzazione chiamata coda** (TCO). Ciò significa che le chiamate di predicato che si trovano in una *posizione di coda* possono essere eseguite nello spazio di stack *costante* se il predicato è deterministico.

L'ottimizzazione della **ricorsione in** coda (TRO) è un caso speciale di ottimizzazione della chiamata di coda.

Leggi Prestazione online: https://riptutorial.com/it/prolog/topic/4205/prestazione

Capitolo 11: Programmazione della logica del vincolo

Examples

CLP (FD)

I vincoli CLP (FD) (*Domini finiti*) implementano l'aritmetica sugli **interi** . Sono disponibili in tutte le implementazioni seria di Prolog.

Esistono due principali casi d'uso dei vincoli CLP (FD):

- Aritmetico intero dichiarativo
- Risoluzione di problemi combinatori come la pianificazione, la pianificazione e le attività di allocazione.

Esempi:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Si noti che se si `is/2` nella seconda query, si verificherà un errore di istanziazione:

```
?- 3 is Y+2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

CLP (Q)

CLP (Q) implementa il ragionamento su numeri *razionali* .

Esempio:

```
?- { 5/6 = X/2 + 1/3 }.
X = 1.
```

CLP (H)

Lo stesso Prolog può essere considerato come **CLP (H)** : Constraint Logic Programming su *termini di Herbrand* . Con questa prospettiva, un programma Prolog posta vincoli su *termini* . Per esempio:

```
?- X = f(Y), Y = a.
X = f(a),
```

```
Y = a.
```

Leggi Programmazione della logica del vincolo online:
https://riptutorial.com/it/prolog/topic/2057/programmazione-della-logica-del-vincolo

Capitolo 12: Programmazione di ordine superiore

Examples

call / N predicates

La famiglia di predicati `call/N` può chiamare obiettivi Prolog arbitrari in fase di esecuzione:

```
?- G=true, call(G).
true.

?- G=(true,false), call(G).
false.
```

maplist / [2,3]

`maplist/2` e `maplist/3` sono predicati di ordine superiore, che consentono di `maplist/3` la definizione di un predicato su un singolo elemento per gli *elenchi* di tali elementi. Questi predicati possono essere definiti usando `call/2` e `call/3` come elementi costitutivi e spediti con molti sistemi Prolog.

Per esempio:

```
?- maplist(dif(a), [X,Y,Z]).
dif(X, a),
dif(Y, a),
dif(Z, a).
```

Meta-chiamata

In Prolog, la cosiddetta **meta-call** è una funzione di linguaggio incorporata. Tutto il codice Prolog è rappresentato dai *termini* Prolog, che consente di costruire gli obiettivi in modo dinamico e di essere utilizzati come altri obiettivi senza ulteriori predicati:

```
?- Goal = dif(X, Y), Goal.
dif(X, Y).
```

Utilizzando questo meccanismo, altri predicati di ordine superiore possono essere definiti in Prolog stesso.

foldl / 4

Una *piega* (da sinistra) è una relazione di ordine superiore tra:

- un predicato con 3 argomenti
- un elenco di elementi

- uno stato iniziale
- uno stato finale, che è il risultato dell'applicazione del predicato a elementi successivi mentre trasporta stati intermedi.

Ad esempio: utilizzare `foldl/4` per esprimere la *somma* di tutti gli elementi in un elenco, utilizzando un predicato come blocco predefinito per definire la somma di *due* elementi:

```
?- foldl(plus, [2,3,4], 0, S).
S = 9.
```

Chiama una lista di obiettivi

Per chiamare un elenco di obiettivi come se fosse una combinazione di obiettivi, combinare la chiamata ai predicati di ordine superiore / 1 e la lista di mappe / 2:

```
?- Gs = [X = a, Y = b], maplist(call, Gs).
Gs = [a=a, b=b],
X = a,
Y = b.
```

Leggi Programmazione di ordine superiore online:
https://riptutorial.com/it/prolog/topic/2420/programmazione-di-ordine-superiore

Capitolo 13: Purezza logica

Examples

dif / 2

Il predicato `dif/2` è un predicato **puro** : può essere utilizzato in tutte le direzioni e con tutti i modelli di istanziazione, *sempre nel* senso che i suoi due argomenti sono *diversi* .

Vincoli CLP (FD)

I vincoli CLP (FD) sono relazioni completamente pure. Possono essere utilizzati in tutte le direzioni per l'aritmetica dichiarativa intera:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Unificazione

L'unificazione è una **pura** relazione. Non produce effetti collaterali e può essere utilizzato in tutte le direzioni, con uno o entrambi gli argomenti completamente o solo parzialmente istanziati.

In Prolog, può avvenire l'unificazione

- **esplicitamente** , usando predicati integrati come `(=)/2` o `unify_with_occurs_check/2`
- **implicitamente** , quando l'unificazione viene utilizzata per selezionare una clausola adatta.

Leggi Purezza logica online: https://riptutorial.com/it/prolog/topic/2058/purezza-logica

Capitolo 14: Ragionamento sui dati

Osservazioni

Una nuova sezione chiamata **Strutture Dati è** stata messa in luce dove sono fornite spiegazioni di alcune strutture + alcuni esempi semplici di creazione. Per mantenere il suo contenuto conciso e senza fronzoli, non dovrebbe contenere alcuna documentazione sulla manipolazione dei dati.

Pertanto, questa sezione è stata rinominata in "Ragionamento sui dati" con lo scopo di generalizzare il ragionamento sui dati in Prolog. Questo potrebbe includere argomenti che vanno da "inferenza top-down" a "attraversamento di liste", così come molti altri. A causa della sua ampia generalizzazione, dovrebbero essere fatte chiare sottosezioni!

Examples

ricorsione

Il prologo non ha iterazione, ma tutta l'iterazione può essere riscritta usando la ricorsione. La ricorsione appare quando un predicato contiene un obiettivo che si riferisce a se stesso. Quando si scrivono tali predicati in Prolog, uno schema ricorsivo standard ha sempre almeno due parti:

- **Clausola base (non ricorsiva)** : in genere le regole del caso base rappresenteranno l'esempio / i più piccoli possibili del problema che si sta tentando di risolvere: un elenco senza membri o un solo membro o se si Lavorando con una struttura ad albero, potrebbe trattarsi di un albero vuoto o di un albero con un solo nodo, ecc. Descrive in modo non ricorsivo la base del processo ricorsivo.
- **Clausola ricorsiva (continua)** : contiene qualsiasi logica necessaria, inclusa una chiamata a se stessa, ricorsione continua.

Ad esempio, definiremo il predicato noto `append/3` . Visto in termini dichiarativi, `append(L1,L2,L3)` vale quando la lista `L3` è il risultato di liste aggiunte `L1` e `L2` . Quando proviamo a capire il significato dichiarativo di un predicato, proviamo a descrivere le soluzioni per le quali il predicato è valido. La difficoltà qui sta nel cercare di evitare ogni dettaglio ricorrente passo dopo passo, tenendo sempre presente il comportamento procedurale che il predicato dovrebbe mostrare.

```
% Base case
append([],L,L).

% Recursive clause
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Il caso base dichiara dichiaratamente che "ogni L aggiunto alla lista vuota è L", nota che questo non dice nulla sul fatto che L sia vuoto - o addirittura che sia una lista (ricorda che in Prolog tutto si riduce a termini):

```
?- append(X,some_term(a,b),Z).
X = [],
Z = some_term(a, b).
```

Per descrivere la regola ricorsiva, sebbene Prolog esegua le regole da sinistra a destra, omettiamo la testa per un secondo e guardiamo prima il corpo - leggendo la regola da destra a sinistra:

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Ora diciamo che se il corpo sostiene: "supponendo che `append(L1,L2,L3)` contenga"

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Quindi lo fa anche la testa: "allora anche `append([X|L1],L2,[X|L3])` "

In inglese semplice ciò si traduce semplicemente in:

> Supponendo che L3 sia la concatenazione di L1 e L2, quindi [X seguito da L3] è anche la concatenazione di [X seguito da L1] e L2.

In un esempio pratico:

> "Supponendo [1,2,3] è la concatenazione di [1] e [2,3], quindi [a, 1,2,3] è anche la concatenazione di [a, 1] e [2,3]. ”

Ora esaminiamo alcune domande:

È sempre consigliabile testare inizialmente il predicato con la **query più generica** anziché fornire un caso di scenario specifico. Pensaci: grazie all'unificazione di Prolog, non siamo tenuti a fornire dati di prova, semplicemente li consegniamo a variabili libere!

```
?- append(L1,L2,L3).
L1 = [],
L2 = L3 ;                                   % Answer #1
L1 = [_G1162],
L3 = [_G1162|L2] ;                          % Answer #2
L1 = [_G1162, _G1168],
L3 = [_G1162, _G1168|L2] ;                  % Answer #3
L1 = [_G1162, _G1168, _G1174],
L3 = [_G1162, _G1168, _G1174|L2] ;          % Answer #4
...
```

Sostituiamo la notazione variabile `_G1162` con lettere alfabetiche per ottenere una panoramica migliore:

```
?- append(L1,L2,L3).
L1 = [],
L2 = L3 ;                                   % Answer #1
L1 = [_A],
L3 = [_A|L2] ;                              % Answer #2
L1 = [_A, _B],
```

```
L3 = [_A, _B|L2] ;                          % Answer #3
L1 = [_A, _B, _C],
L3 = [_A, _B, _C|L2] ;                      % Answer #4
...
```

Nella prima risposta, il caso base è stato abbinato al modello e il Prolog ha istanziato `L1` alla lista vuota e `L2` e `L3` unificati dimostrando che `L3` è la concatenazione della lista vuota e L2.

Alla risposta n. 2, attraverso il backtrack cronologico, la clausola ricorsiva entra in gioco e Prolog prova a dimostrare che alcuni elementi nella testa di `L1` concatenati con `L2` sono `L3` con lo stesso elemento nella sua testa di lista. Per fare ciò, una nuova variabile libera `_A` è unificata con la testa di L1 e L3 ha dimostrato di essere ora `[_A|L2]` .

Viene creata una nuova chiamata ricorsiva, ora con `L1 = [_A]` . Ancora una volta, Prolog prova a dimostrare che alcuni elementi posizionati nella testa di `L1` , concatenati con `L2` sono `L3` con lo stesso elemento nella sua testa. Notare che `_A` è già la testa di `L1` , che corrisponde perfettamente alla regola, quindi ora, attraverso la ricorsione, Prolog mette `_A` davanti a una nuova variabile libera e otteniamo `L1 = [_A,_B]` e `L3 = [_A,_B|L2]`

Vediamo chiaramente che il pattern ricorsivo si ripete e può facilmente vedere che, ad esempio, il risultato del 100 ° step in ricorsione sarà simile a:

```
L1 = [X1,X2,..,X99],
L3 = [X1,X2,..,X99|L2]
```

Nota: come è tipico per il buon codice Prolog, la definizione ricorsiva di `append/3` ci fornisce non solo la possibilità di *verificare* se una lista è la concatenazione di altre due liste, ma *genera* anche tutte le risposte possibili soddisfacendo le relazioni logiche con o elenchi parzialmente istanziati.

Accedere agli elenchi

Membro

`member/2` ha un `member(?Elem, ?List)` firma `member(?Elem, ?List)` e indica `true` se `Elem` è un membro di `List` . Questo predicato può essere utilizzato per accedere a variabili in un elenco, in cui vengono recuperate diverse soluzioni tramite il backtracking.

Query di esempio:

```
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

?- member(X,[Y]).
X = Y.

?- member(X,Y).
Y = [X|_G969] ;
Y = [_G968, X|_G972] ;
Y = [_G968, _G971, X|_G975] ;
```

```
Y = [_G968, _G971, _G974, X|_G978]
...
```

Pattern matching

Quando gli indici a cui devi accedere sono piccoli, la corrispondenza dei modelli può essere una buona soluzione, ad esempio:

```
third([_,_,X|_], X).
fourth([_,_,_,X|_], X).
```

Leggi Ragionamento sui dati online: https://riptutorial.com/it/prolog/topic/2005/ragionamento-sui-dati

Capitolo 15: Strutture dati

Examples

elenchi

Le liste sono un tipo speciale di *termine composto* . Le liste sono definite in modo induttivo:

- l'atomo `[]` è una lista, che indica la *lista vuota* .
- *se* `Ls` è una lista, allora il termine `'.'(L, Ls)` è *anche* una lista.

C'è una sintassi speciale per denotare gli elenchi comodamente in Prolog:

1. La lista `'.'(a, '.'(b, '.'(c, [])))` può anche essere scritta come `[a,b,c]` .
2. Il termine `'.'(L, Ls)` può anche essere scritto come `[L|Ls]` .

Queste annotazioni possono essere combinate in qualsiasi modo. Ad esempio, il termine `[a,b|Ls]` è una lista *se* `Ls` è una lista.

Creare liste

Una lista di letterali unificati con la variabile List:

```
?- List = [1,2,3,4].
List = [1, 2, 3, 4].
```

Costruire una lista consing:

```
?- Tail = [2, 3, 4], List = [1|Tail].
Tail = [2, 3, 4],
List = [1, 2, 3, 4].
```

Costruire un elenco di valori sconosciuti utilizzando la `length/2` integrata `length/2` :

```
?- length(List,5).
List = [_G496, _G499, _G502, _G505, _G508].
```

Poiché in Prolog tutto è essenzialmente un Termine, le liste si comportano in modo eterogeneo:

```
?- List = [1, 2>1, this, term(X), 7.3, a-A].
List = [1, 2>1, this, term(X), 7.3, a-A].
```

Ciò significa che un elenco può contenere anche altri elenchi, chiamati anche elenchi interni:

```
List = [[1,2],[3,[4]]].
```

Pairs

Per convenzione, il functor `(-)/2` è spesso usato per denotare **coppie** di elementi in Prolog. Ad esempio, il termine `-(A, B)` indica la coppia di elementi `A` e `B` In Prolog, `(-)/2` è definito come *operatore infisso* . Pertanto, il termine può essere scritto in modo equivalente come `AB` .

Molti predicati comunemente disponibili utilizzano anche questa sintassi per indicare coppie. Esempi di questo sono `keysort/2` e `pairs_keys_values/3` .

Liste di associazioni

In tutti i seri sistemi Prolog, sono disponibili **elenchi di associazioni** che consentono un accesso più rapido e lineare a una raccolta di elementi. Questi elenchi di associazioni si basano in genere su *alberi bilanciati* come gli **alberi AVL** . Esiste una libreria di dominio pubblico chiamata `library(assoc)` che viene fornita con molti sistemi Prolog e che fornisce operazioni *O (log (N))* per l'inserimento, il recupero e la modifica di elementi in una raccolta.

condizioni

A un livello molto alto, Prolog ha solo un singolo tipo di dati, chiamato **termine** . In Prolog, tutti i dati sono rappresentati dai termini di Prolog. I termini sono definiti in modo induttivo:

- un **atomo** è un termine. Esempi di atomi sono: `x` , `test` e `'quotes and space'` .
- una **variabile** è un termine. Le variabili iniziano con una lettera maiuscola o underscore `_` .
- numeri interi e numeri in virgola mobile sono termini. Esempi: `42` e `42.42` .
- un **termine composto** è un termine, definito induttivamente come segue: *Se* `T1` , `T2` , ..., `T_n` sono termini, *quindi F* (`T1` , `T2` , ..., `T_n`) è anche un termine, dove *F* è chiamato il **funtore** di il termine composto.

Termini con campi denominati utilizzando la libreria (record)

La libreria `[record][1]` offre la possibilità di creare termini composti con campi denominati. La direttiva `:- record/1 <spec>` compila in una raccolta di predicati che inizializzano, impostano e ottengono campi nel termine definito da `<spec>` .

Ad esempio, possiamo definire una struttura di dati `point` con campi denominati `x` e `y` :

```
:- use_module(library(record)).

:- record point(x:integer=0,
                y:integer=0).

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

?- default_point(Point), point_x(Point, X), set_x_of_point(10, Point, Point1).
Point = point(0, 0),
X = 0,
Point1 = point(10, 0).

?- make_point([y(20)], Point).
Point = point(0, 20).

?-  is_point(X).
```

```
false.

?- is_point(point(_, _)).
false.

?- is_point(point(1, a)).
false.

?- is_point(point(1, 1)).
true.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

Leggi Strutture dati online: https://riptutorial.com/it/prolog/topic/2417/strutture-dati

Capitolo 16: Strutture di controllo

Examples

Disgiunzione (OR logico), implicita contro esplicita

Prolog prova le clausole alternative per un predicato nell'ordine di apparizione:

```
likes(alice, music).
likes(bob, hiking).

// Either alice likes music, or bob likes hiking will succeed.
```

L'operatore di disgiunzione (OR) `;` può essere usato per esprimere questo in una regola:

```
likes(P,Q) :-
    ( P = alice , Q = music ) ; ( P = bob , Q = hiking ).
```

Le parentesi sono importanti qui per chiarezza. Vedere questa domanda la precedenza rispetto per la collaborazione `,` e la disgiunzione `;`.

Congiunzione (logico AND)

La congiunzione (AND logico) è rappresentata dalla virgola `,` operatore (tra gli altri ruoli).

La congiunzione tra clausole può apparire in una query:

```
?- X = 1, Y = 2.
```

Congiunzione può anche apparire tra le clausole subgoal nel corpo di una regola:

```
triangleSides(X,Y,Z) :-
    X + Y > Z, X + Z > Y, Y + Z > X.
```

Taglia (rimuovi punti di scelta)

A volte è auspicabile impedire a Prolog di tornare indietro in soluzioni alternative. Lo strumento di base a disposizione del programmatore per impedire al prologo di continuare a tornare nel suo backtrack è l'operatore di taglio. considerare quanto segue.

```
% (percent signs mean comments)
% a is the parent of b, c, and d.
parent(a,b).
parent(a,c).
parent(a,d).
```

Qui il predicato `parent/2` succede più di una volta quando

```
?- parent(a,X).
```

è chiamato. Per evitare che il prologo cerchi altre soluzioni dopo che il primo è stato trovato, si utilizzerà l'operatore di taglio, in questo modo.

```
?- parent(a,X), !.
```

Questo avrà X uguale a b (come è la prima soluzione possibile) e non cercherò più soluzioni.

Leggi Strutture di controllo online: https://riptutorial.com/it/prolog/topic/4479/strutture-di-controllo

Capitolo 17: Utilizzando Modern Prolog

Examples

introduzione

Molti dei moderni sistemi Prolog sono in continuo sviluppo e hanno aggiunto nuove funzionalità per affrontare le carenze classiche della lingua. Sfortunatamente, molti libri di testo di Prolog e persino i corsi di insegnamento presentano ancora solo il prologo superato. Questo argomento ha lo scopo di illustrare come il Prolog moderno ha superato alcuni dei problemi e la sintassi piuttosto crufty che appare nel vecchio Prolog e potrebbe ancora essere introdotta.

CLP (FD) per l'aritmetica dei numeri interi

Tradizionalmente, Prolog eseguiva l'aritmetica usando gli operatori `is` e `=:=` . Tuttavia, diversi Prolog attuali offrono CLP (FD) (Constraint Logic Programming su Finite Domains) come alternativa più pulita per l'aritmetica dei numeri interi. CLP (FD) si basa sulla memorizzazione dei vincoli che si applicano a un valore intero e sulla combinazione di quelli in memoria.

CLP (FD) è un'estensione nella maggior parte dei Prolog che lo supportano, quindi deve essere caricato in modo esplicito. Una volta caricato, la sintassi `#=` può prendere il posto di entrambi `is` e `=:=` . Ad esempio, in SWI-Prolog:

```
?- X is 2+2.
X = 4.

?- use_module(library(clpfd)).
?- X #= 2+2.
X = 4.
```

A differenza di `is` , `#=` è in grado di risolvere equazioni semplici e unificare in entrambe le direzioni:

```
?- 4 is 2+X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- 4 #= 2+X.
X = 2.
```

CLP (FD) fornisce la propria sintassi del generatore.

```
?- between(1,100,X).
X = 1;
X = 2;
X = 3...

?- X in 1..100.
X in 1..100.
```

Si noti che il generatore non viene effettivamente eseguito: viene memorizzato solo il vincolo dell'intervallo, pronto per essere combinato con vincoli successivi. Il generatore può essere forzato a eseguire (e vincoli di forza bruta) usando il predicato `label` :

```
?- X in 1..100, label([X]).
X = 1;
X = 2;
X = 3..
```

L'uso di CLP può consentire una riduzione intelligente dei casi di forza bruta. Ad esempio, utilizzando l'aritmetica dei numeri interi vecchio stile:

```
?- trace.
?- between(1,10,X), Y is X+5, Y>10.
...
Exit: (8) 6 is 1+5 ? creep
Call: (8) 6 > 10 ? creep
...
X = 6, Y = 11; ...
```

Il Prolog circola ancora sui valori 1-5 anche se è matematicamente dimostrabile dalle condizioni date che questi valori non possono essere utili. Utilizzando CLP (FD):

```
?- X in 1..10, Y #= X+5, Y #> 10.
X is 6..10,
X+5 #= Y,
Y is 11..15.
```

CLP (FD) esegue immediatamente i calcoli matematici e risolve gli intervalli disponibili. L'aggiunta di `label([Y])` farà sì che X esegua il loop solo attraverso i valori utili 6..10. In questo esempio di giocattolo, questo non aumenta le prestazioni perché con un intervallo così piccolo come 1-10, l'elaborazione algebra impiega tutto il tempo che il ciclo avrebbe fatto; ma quando viene elaborato un numero maggiore di numeri, ciò può ridurre sensibilmente il tempo di calcolo.

Il supporto per CLP (FD) è variabile tra Prologs. Il migliore sviluppo riconosciuto di CLP (FD) è in SICStus Prolog, che è commerciale e costoso. SWI-Prolog e altri Prolog aperti spesso hanno qualche implementazione. Visual Prolog non include CLP (FD) nella sua libreria standard, sebbene siano disponibili librerie di estensioni per esso.

Forall invece dei loop pilotati da errori

Alcuni manuali "classici" di Prolog utilizzano ancora la sintassi del ciclo guidata da errori e soggetta a errori, in cui viene utilizzato un costrutto di `fail` per forzare il backtracking ad applicare un obiettivo a ogni valore di un generatore. Ad esempio, per stampare tutti i numeri fino a un determinato limite:

```
fdl(X) :- between(1,X,Y), print(Y), fail.
fdl(_).
```

La stragrande maggioranza dei Modern Prologs non richiede più questa sintassi, ma fornisce un

predicato di ordine superiore per risolvere questo problema.

```
nicer(X) :- forall(between(1,X,Y), print(Y)).
```

Non solo è molto più facile da leggere, ma se un obiettivo che poteva fallire veniva usato al posto della *stampa* , il suo errore sarebbe stato rilevato e trasmesso correttamente - mentre i guasti degli obiettivi in un ciclo guidato da errori sono confusi con l'errore forzato che guida il ciclo.

Visual Prolog ha uno zucchero sintattico personalizzato per questi loop, combinato con i predicati di funzione (vedi sotto):

```
vploop(X) :- foreach Y = std::fromTo(1,X) do
                 console::write(X)
             end foreach.
```

Anche se questo sembra un imperativo *per il* ciclo, segue ancora le regole Prolog: in particolare, ogni iterazione del *foreach* è il suo ambito.

Predicati di stile funzionale

Tradizionalmente in Prolog, le "funzioni" (con un output e input associati) venivano scritte come predicati regolari:

```
mangle(X,Y) :- Y is (X*5)+2.
```

Questo può creare la difficoltà che, se un predicato stile-funzione viene chiamato più volte, è necessario impostare variabili temporanee "a catena".

```
multimangle(X,Y) :- mangle(X,A), mangle(A,B), mangle(B,Y).
```

Nella maggior parte dei Prolog, è possibile evitare ciò scrivendo un operatore infisso alternativo da utilizzare al posto di `is` che espande le espressioni compresa la funzione alternativa.

```
% Define the new infix operator
:- op(900, xfy, <-).

% Define our function in terms of the infix operator - note the cut to avoid
% the choice falling through
R <- mangle(X) :- R is (X*5)+2, !.

% To make the new operator compatible with is..
R <- X :-
    compound(X),            % If the input is a compound/function
    X =.. [OP, X2, X3],     % Deconstruct it
    R2 <- X2,               % Recurse to evaluate the arguments
    R3 <- X3,
    Expr =.. [OP, R2, R3],  % Rebuild a compound with the evaluated arguments
    R is Expr,              % And send it to is
    !.
R <- X :- R is X, !.        % If it's not a compound, just use is directly
```

Ora possiamo scrivere:

```
multimangle(X,Y) :- X <- mangle(mangle(mangle(Y))).
```

Tuttavia, alcuni moderni Prolog vanno oltre e offrono una sintassi personalizzata per questo tipo di predicato. Ad esempio, in Visual Prolog:

```
mangle(X) = Y :- Y = ((X*5)+2).
multimangle(X,Y) :- Y = mangle(mangle(mangle(X))).
```

Si noti che l'operatore `<-` e il predicato dello stile funzionale di cui sopra si comportano ancora come *relazioni* - è legale per loro avere punti di scelta ed eseguire più unificazione. Nel primo esempio, evitiamo questo usando tagli. In Visual Prolog, è normale utilizzare la sintassi funzionale per le relazioni ei punti di scelta vengono creati nel modo normale - ad esempio, l'obiettivo `X = (std::fromTo(1,10))*10` riesce con i binding X = 10 , X = 20, X = 30, X = 40, ecc.

Dichiarazioni flusso / modalità

Quando si programma in Prolog non è sempre possibile, o desiderabile, creare predicati che unificano per ogni possibile combinazione di parametri. Ad esempio, il predicato `between(X,Y,Z)` che esprime che Z è numericamente tra X e Y. È facilmente implementabile nei casi in cui X, Y e Z sono tutti vincolati (o Z è tra X e Y o non lo è), o dove X e Y sono vincolati e Z è libero (Z si unifica con tutti i numeri tra X e Y, o il predicato fallisce se Y <X); ma in altri casi, ad esempio dove X e Z sono legati e Y è libero, esiste potenzialmente un numero infinito di unificazioni. Sebbene questo possa essere implementato, di solito non lo sarebbe.

Dichiarazione di flusso o *dichiarazioni di modalità* consentono una descrizione esplicita di come si comportano i predicati quando vengono chiamati con diverse combinazioni di parametri associati. Nel caso di `between` , la dichiarazione sarebbe:

```
%! between(+X,+Y,+Z) is semidet.
%! between(+X,+Y,-Z) is nondet.
```

Ogni linea specifica un potenziale modello di chiamata per il predicato. Ogni argomento è decorato con `+` per indicare i casi in cui è vincolato, o `-` per indicare i casi in cui non lo è (ci sono anche altre decorazioni disponibili per tipi più complessi come tuple o liste che possono essere parzialmente rilegate). La parola dopo *è* indica il comportamento del predicato in questo caso, e può essere uno dei seguenti:

- `det` se il predicato riesce sempre senza punto di scelta. Ad esempio `add(+X,+Y,-Z)` è `det` perché aggiungendo due numeri X e Y avrà sempre esattamente una risposta.
- `semidet` se il predicato ha esito positivo o negativo, senza alcun punto di scelta. Come sopra, `between(+X,+Y,+Z)` è `semidet` perché Z è tra X e Y oppure non lo è.
- `multi` se il predicato ha sempre successo, ma può avere punti di scelta (ma potrebbe anche non esserlo). Ad esempio, `factor(+X,-Y)` sarebbe `multi` perché un numero ha sempre almeno un fattore - se stesso - ma potrebbe avere di più.
- `nondet` se il predicato può avere successo con punti di scelta o fallire. Ad esempio, `between(+X,+Y,-Z)`

è `nondet` perché potrebbero esserci diverse possibili unificazioni di Z ai numeri tra X e Y, oppure se Y <X allora non ci sono numeri tra loro e il predicato fallisce.

Le dichiarazioni di flusso / modalità possono anche essere combinate con l'etichettatura degli argomenti per chiarire cosa significano i termini o con la digitazione. Ad esempio,
`between(+From:Int, +To:Int, +Mid:Int) is semidet .`

Nei puri Prolog, le dichiarazioni di flusso e modalità sono facoltative e utilizzate solo per la generazione di documentazione, ma possono essere estremamente utili per aiutare i programmatori a identificare la causa degli errori di istanziazione.

In Mercury, le dichiarazioni e i tipi di flusso e modalità sono obbligatori e vengono convalidati dal compilatore. La sintassi utilizzata è come sopra.

In Visual Prolog anche le dichiarazioni e i tipi di flusso e modalità sono obbligatori e la sintassi è diversa. La dichiarazione di cui sopra sarebbe scritta come:

```
between : (int From, int To, int Mid) determ (i,i,i) nondeterm (i,i,o).
```

Il significato è lo stesso di sopra, ma con le differenze che:

- Le dichiarazioni di flusso / modalità sono separate dalle dichiarazioni di tipo (poiché si presume che il flusso / la modalità per un singolo predicato non varieranno con il sovraccarico di tipo);
- `i` e `o` sono usati per `+` e `-` e sono abbinati ai parametri basati sull'ordinamento;
- I termini usati sono diversi. `det` diventa `procedure` , `semidet` diventa `determ` e `nondet` diventa `nondeterm` (`multi` è ancora `multi`).

Leggi Utilizzando Modern Prolog online: https://riptutorial.com/it/prolog/topic/5499/utilizzando-modern-prolog

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Prolog Language	coder, Community, Eyal, Limmen, manlio, mat, Nick the coder, Norman Creaney, Rajat Jain, Ruslan López Carro, SeekAndDestroy, Willem Van Onsem, Xevaquor
2	Alberi di derivazione	SeekAndDestroy
3	Elenchi di differenze	ZenLulz
4	Gestione degli errori ed eccezioni	mat
5	Grammatiche di clausole definite (DCG)	false, mat, Will Ness, ZenLulz
6	Linee guida per la codifica	mat
7	Monotonicità	manlio, mat
8	operatori	false, mat, SeekAndDestroy
9	Predicati extra logici	false, mat
10	Prestazione	mat
11	Programmazione della logica del vincolo	mat, SeekAndDestroy
12	Programmazione di ordine superiore	4444, Eyal, mat
13	Purezza logica	mat
14	Ragionamento sui dati	Daniel Lyons, manlio, mat, SeekAndDestroy
15	Strutture dati	Eyal, mat, SeekAndDestroy
16	Strutture di controllo	hardmath, Nick the coder
17	Utilizzando Modern	Mark Green

	Prolog	