## Regole generali di Prolog

Il compilatore prolog usa due importanti regole:

1. **Computation rule**: nel corpo di una regola, sceglie in ordine le clausole da eseguire
2. **Search rule**: sceglie in modo ordinato le clausole così come sono scritte

Un programma prolog è composto da fatti, regole e procedure:

1. In una procedura prima i fatti poi le regole (prima le cose semplice poi le complicate)
2. Nel corpo di una regola, prima i goal più semplici da dimostrare o refutare, poi quelli più difficili.

Le strutture si creano con un **funtore**: f(t1,...,tn): crea un albero con n figli.

Una **lista** è un albero completamente sbilanciato a sinistra. È una struttura creata con un funtore, tramite l'operatore prefisso '.' o infisso '|':

1. [a,b,c] = .(a,.(b,.(c,[]))).
2. [a,b,c] = [a | [ b | [ c | [] ] ] ].

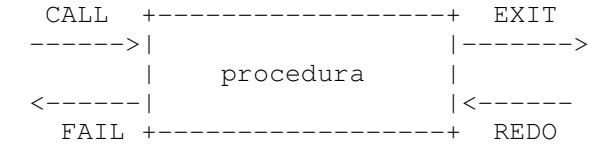
Le stringhe sono liste di caratteri ascii:

- “abc” = [97, 98, 99]

Il **not** è la negazione come fallimento. Se goal fallisce allora not(goal) ha successo e viceversa. *Not non è la negazione logica*: se not(Goal) ha successo non significa che la negazione di Goal è dimostrabile dal programma, ma semplicemente che Goal non è dimostrabile.

## Il flusso di controllo

Secondo il **modello a scatole**, ogni invocazione di una procedura viene rappresentata con una scatola con quattro porte:




Attraverso le porte è possibile seguire il flusso di controllo nell'esecuzione di un programma. In particolare, il flusso può essere pensato come un puntatore che si muove lungo il codice all'interno della scatola entrando per le porte di ingresso e uscendo per le porte di uscita.

- La porta **CALL** indica il punto di ingresso nella scatola all'atto dell'invocazione iniziale della procedura. Dopo l'ingresso da questa porta il puntatore si muove lungo il codice della procedura seguendo l'ordine con cui si tenta di soddisfare i singoli subgoal.
- La porta **EXIT** indica il punto di uscita dalla scatola dopo un ritorno con successo dalla procedura.
- La porta **REDO** indica il punto di ingresso nella scatola per la ricerca di una soluzione alternativa a seguito di un operazione di backtracking. Ciò vuol dire un tentativo di soddisfare in modo diverso uno dei subgoal nel corpo della clausola che aveva avuto successo prima. Dopo l'ingresso da questa porta, il puntatore si muove all'indietro lungo il codice a partire dal punto in cui aveva lasciato la scatola attraverso la porta **EXIT** nel

tentativo di trovare una nuova soluzione per un subgoal. Se non ci sono soluzioni alternative, il puntatore passa a una nuova clausola.

- La porta **FAIL** indica il punto di uscita dalla scatola nel caso in cui l'invocazione non ha successo. Il puntatore ritorna al codice da cui era stata invocata la procedura, muovendosi all'indietro alla ricerca di soluzioni alternative.

Per ogni invocazione vi è sempre un solo ingresso del puntatore attraverso la porta **CALL** e una sola uscita attaverso la porta **FAIL**, mentre il puntatore può entrare e uscire diverse volte attraverso le porte **REDO** e **EXIT**, rispettivamente, per effetto del backtracking.

E` opportuno indicare ogni scatola, e quindi ogni invocazione, con un numero diverso. Ciò è utile quando, soprattutto a causa di definizioni ricorsive, possono esserci più invocazioni della stessa procedura.

Le scatole vengono collegate fra loro nel modo seguente:

```
FAIL --> REDO
FAIL --> CALL
EXIT --> CALL
```

Questi collegamenti determinano lo scorrere del flusso in due direzioni: una in avanti e una all'indietro. Il movimento all'indietro corrisponde a operazioni di **backtracking**.

La presenza del **cut**, interrompe il collegamento **FAIL - REDO** corrispondente e ridirige l'uscita **FAIL** all'uscita **FAIL** della porta più esterna. Il flusso può essere seguito con un puntatore lungo le frecce che collegano le scatole. L'ingresso attraverso una porta **REDO** provoca un movimento del puntatore all'indietro lungo le scatole attraverso le connessioni **FAIL - REDO** nel tentativo di soddisfare in modo diverso un subgoal. Un successo fa uscire il puntatore dalla porta **EXIT** ripristinando il movimento in avanti. Il movimento del puntatore si alterna in avanti e all'indietro, a seconda della porta da cui viene fatto uscire, finchè questo non esce definitivamente da una delle porte più esterne **EXIT** o **FAIL**.

La biforcazione della freccia **REDO** indica i due punti di ritorno in un'operazione di backtracking. Come regola, si ritorna alla scatola da cui si è usciti l'ultima volta attraverso la porta **EXIT**.

Se il cut non cambia il significato del programma, ho un **cut verde**: migliora l'efficienza; in caso di fallimento non si tentano altre alternative, che sarebbero comunque destinate a fallire.

Se il cut viene utilizzato per ottenere il significato corretto del programma, ho un **cut rosso**. Il cut rosso dice al sistema: “se sei arrivato fin qui, hai trovato e scelto la regola corretta per soddisfare questo goal (non fare backtracking se fallisci, perché questa è l'unica regola corretta!)”

**Il cut può rovinare la reversibilità dei predicati**: utilizzando il cut si deve aver presente come si useranno esattamente le regole!

# Operatori – funzioni

## Unificazione

- **=** unifica : f(X, pippo) = f(pluto, Y)
- **unify_with_occurs_check(X,f(X))**: esegue l'occur check

## Classificazione dei termini

- **var(term)**:  verifica se term è una variabile non vincolata (var(padre(X)) non lo è)
- **atom(term)**: verifica se term è un atomo
- **atomic(term)**: verifica se term è un atomo, una stringa, un numero intero o decimale

## Funtori

- **functor(term,functor,arity)**: riporta il nome della funzione e l'arità associata alla descrizione term data (functor(f(a,b,c),F,N) dà F=f e N=3)
- **arg(arg,term,value)**: riporta in value l'argomento arg richiesto della funzione term.
- f(a,b,c) **=..** X --> X = [f, a, b, c]

## Liste

- **member(elem,list)**: riporta se elem è un elemento di list
- **append(list1,list2,list3)**: concatena in list3 le liste list1 e list2
- **length(list,int)**: riporta in int la lunghezza di list
- **delete(list1,elem,list2):** elimina tutti gli elem di list1 e riporta il risultato in list2
- **flatten(list1,list2)**: trasforma una list1 in una lista flat, con un solo livello.

## Flusso di controllo

- **!:** l'operatore cut porta all'uscita fail della porta più esterna interrompendo il collegamento fail-redo
- **fail**: invoca immediatamente il fail. Forza quindi il redo tramite il collegamento fail-redo.

## Manipolazione base di dati

- **dynamic(func/arity)**: dichiara che la func di arità arity ha contenuto dinamico
- **asserta(term)**: il term è inserito in cima ai fatti
- **assertz(term)**: il term è inserito in coda ai fatti
- **retract(term)**: rimuove il term dal database

## Query del secondo ordine

- **bagof(template,goal,bag):** restituisce in bag le alternative derivanti dalla variabile template alle possibili interpretazioni di goal, che possiede la variabile template vincolata: bagof(C, foo(A,B,C), Bag).
- **setof(template,goal,set):** equivalente a bagof, ma restituisce lista ordinate e senza ripetizioni
- **findall(template,goal,bag)**: equivalente a bagof, ma restituisce in una lista tutte le possibilità, e se non esiste alcun template che rende vero il goal l'esecuzione non fallisce come bagof, ma genera una lista vuota.