Progetti tipo C

I tipi

```
type stato.          % gli stati mondo-agente
type decisione.   % le decisioni possibili dell'agente
type info_inizio. % informazioni da fornire all'inizio storia
type assumibile.  % metatipo per i predicati assumibili
type [  %%  INIZIANO DECISIONI
        impossibile(decisione),
        eseguita(decisione),
        fallita(decisione, list(decisione)),
        inizio_storia(info_inizio),

        %% EVENTI INTERMEDI, NON INIZIANO DECISIONI
        deciso(decisione),
        pianificata(decisione, list(decisione)),
        transizione(stato, decisione, stato),
        stop(stato,list(evento)) ]: evento.
type [ {assumibile}, non(assunzione)]: assunzione.
```

Partenza

```
start(Avvio) :-
        stato_iniziale(S0,Avvio),      % stato iniziale dal progetto
        ricorda(S0,[],inizio_storia(Avvio), Storia0),
        vai(S0, Storia0).
```

Il ciclo decidi-pianifica-esegui

```
vai(Stato, Storia) :-
    decidi(Stato, Storia, Decisione),
    (       fine(Decisione),!  % l'agente ha deciso di finire
            mostra_fine(Stato,Decisione,Storia)
     ;      % l'agente non ha deciso di finire e pianifica e attua la decisione
            (   pianifica(Stato, Storia, Decisione, Piano) ->
                %  vi è un piano di attuazione della Decisione;
                 ricorda(Stato, Storia, pianificata(Decisione,Piano), PStoria),
                 esegui(Stato, PStoria, Decisione, Piano, NStato, NStoria)
            ;   %  non vi è un piano di attuazione della decisione;
                %  l'agente non cambia stato e ricorda l'impossibilita'
                 NStato=Stato,
                 ricorda(Stato, Storia, impossibile(Decisione), NStoria)
            ), !,
     % e la storia prosegue
     vai(NStato, NStoria)    ).
```

Il ciclo esegui

```
esegui(Stato,Storia, Decisione, [A|Piano], NStato, NStoria) :-
        esegui_azione(Stato, Storia, A, AStato), !,
        ricorda(Stato, Storia, transizione(Stato, A, AStato), AStoria),
        esegui(AStato,AStoria,Decisione,Piano,NStato,NStoria).

esegui(Stato,Storia,Decisione, [A|Piano], Stato, NStoria) :-
        ricorda(Stato, Storia, fallita(Decisione, [A|Piano]), NStoria).

esegui(Stato,Storia,Decisione,[],Stato,NStoria) :-
        ricorda(Stato, Storia, eseguita(Decisione), NStoria).
```

Ricordare e pianificare

```
ricorda(Stato, Storia,Evento, [Evento|Storia]) :-
        %  quando l'agente ricorda un evento aggiorna anche la base di conoscenza dinamica
        aggiorna_conoscenza(Stato,Storia,Evento).

pianifica(_S,_Storia,Decisione,[Decisione]) :-
        azione(Decisione), !.

pianifica(S,Storia,Decisione,Piano) :-
        piano(S,Storia, Decisione,Piano).
```

Imparare

```
:- dynamic(conosce/1).
:- dynamic(assunto/1).


%%%   B1) Acquisizione nuova conoscenza e revisione della vecchia

impara(A) :-
      forall(  assunto(Ass),   (inconsistenti(Ass,A) -> retract(assunto(Ass)) ; true)),
     assert(conosce(A)).

inconsistenti(A1,A2) :-
            A1=non(A2), !   ;    A2=non(A1), ! ;  contraria(A1,A2).
```

Pensare

```
pensa(_Info, true, Ass, Ass) :- !.

pensa(Info, (A,B), Ass1, Ass2) :- !,
          pensa(Info, A,Ass1, Ass),
          pensa(Info, B,Ass,Ass2).

pensa(Info, A,Ass1,Ass2) :-
          assumibile(A),!,
          assume_o_conosce(Info, A,Ass1,Ass2).

pensa(Info, A,Ass1,Ass2) :-
          meta(A),!,
          clause(A,Body),
           pensa(Info, Body,Ass1,Ass2).

pensa(_Info, A,Ass,Ass) :- call(A).
```

```
assume_o_conosce(_Info, A, Ass,Ass) :-
      conosce(A), !.
assume_o_conosce(_Info, A, Ass,[A|Ass]) :-
      assunto(A), !.
assume_o_conosce(Info, non(A), Ass, [non(A)|Ass]) :- !,
     not(conosce(A)),
     decide_se_assumere(Info,non(A)),
     check_ground(non(A)).
assume_o_conosce(Info, A, Ass, [A|Ass]) :-
     not(conosce(non(A))),
     decide_se_assumere(Info, A),
     check_ground(A).
```

PIANIFICARE

- Usare A* o altro
- In fase di pianificazione usare pensa(…..) che usa conosce ed assume
- In fase di esecuzione usare la base dati dinamica che rappresenta il mondo
- Durante l’ esecuzione imparare con aggiorna_conoscenza (impara passando da assume a conosce)

Lanciare una storia

E’ stata introdotta una rudimentale interfaccia utente, richiamabile con help_progetto. Si tratta di lanciare vai(I), dove I è un indice intero che identifica una mappa su cui l’agente si muoverà. Attualmente sono caricate le mappe 1,2,3 (vai(1), vai(2), vai(3) risultano eseguibili).

Per debuggare

attiva_debug.
disattiva_debug.

si possono riscrivere i predicati mostra_... per personalizzare, vedi di seguito

APPENDICE: VISUALIZZAZIONI DI DEFAULT USATE IN ASSENZA DI QUELLE PERSONALIZZATE

Distinguiamo fra:

- Visualizzazione di debug: attivata con «attiva_debug», «disattiva_debug», è da usare per debuggare il progetto, esaminando i vari passaggi eseguiti da vai.pl con il vostro progetto;
- Visualizzazione di simulazione o «filmato»: sostituisce quella che potrebbe essere un filmato di animazione; semplicemente, stampa la sequenza degli stati percorsa dall'agente; la stampa è passo-passo, premendo <RETURN> si avanza ogni volta di un passo.

Le visualizzazioni possono essere personalizzate; si raccomanda di personalizzare la visualizzazione del «filmato» per rendere leggibile il comportamento dell'agente. Di seguito trovate come operate per personalizzare le visualizzazioni, sia per la fase di debug, sia per il «filmato».

Modificare le visualizzazioni di default della fase di debugging

- In vai, le chiamate alle visualizzazioni sono implementate come segue:

  catch(<predicato personalizzato>, _, <predicato di default>)

  in questo modo, se voi non avete definito il predicato personalizzato, la chiamata solleva l'eccezione di «predicato non definito», che viene catturata mandando in esecuzione il predicato di default. Ad esempio:

```
catch(mostra_conoscenza,_,d_mostra_conoscenza),
    % se voi non definite mostra_conoscenza, viene eseguita
    % d_mostra_conoscenza  (d sta per default)
```

TROVATE i predicati da definire in modo personalizzato nel punto C) di vai_if.pl

MODIFICARE il «filmato»

- In vai il «filmato» del comportamento dell'agente per default è simulato stampando la storia percorsa dall'agente stato di conoscenza per stato di conoscenza.
- Ciò avviene con i predicati **mostra_start** e **mostra_transizione**, riportati nella slide che segue, che usano come default delle writeln per mostrare stato e transizione e **d_mostra_conoscenza** per mostrare la conoscenza dell'agente
- E' possibile personalizzare la visualizzazione definendo **mostra_conoscenza** e **mostra_transizione_stato** e **mostra stato iniziale** o ridefinendo gli interi mostra_statrt e mostra_transizione
  - Nel progetto tipo è stato personalizzato solo **mostra_conoscenza**, che mostra la mappa corrispondente allo stato di conoscenza dell'agente
  - La definizione personalizzata di mostra_conoscenza ha effetto sia sulla fase di simulazione, sia sulla fase di debugging

```
mostra_start(S) :-
        not(debug_on),!,
        catch(mostra_stato_iniziale(S), _, writeln('START nello stato':S)),
        %  se non definite mostra_stato_iniziale(S), viene stampato 'START nello stato':S
        catch(mostra_conoscenza,_,d_mostra_conoscenza),
         %  se non definite mostra_conoscenza, viene eseguita d_mostra_conoscenza
        ask('<RETURN> per proseguire').
mostra_start(_).


mostra_transizione(S1,A,S2) :-
       not(debug_on),!,  %avviene in fase di simulazione e non di debuggin
       catch(mostra_transizione_stato(S1,A,S2), writeln('ESEGUITA ':S1->A->S2)),
        %  se non definite mostra_transizione_stato(S1,A,S2), viene stampato 'ESEGUITA ':S1->A->S2))
        catch(mostra_conoscenza,_,d_mostra_conoscenza),
        ask('<RETURN> per proseguire').
mostra_transizione(_S1,_A,_S2).  %in fase di debug ha successo senza far nulla
```