Event Programming in Prolog

- Events in our quagent Prolog API are just *list of strings*

```prolog
%%% print an event list
print_events([]) :- nl.
print_events([H|T]) :-
    writeln(H),
    print_events(T).
```

Compared to:

```prolog
:- use_module(library(porter_stem)).

%%% print an event list
print_events([]) :- nl.
print_events([H|T]) :-
    tokenize_atom(H,TokenList),  % tokenize the string H
    writeln(TokenList),
    print_events(T).
```
%% libraries and modules
:- consult('quagent.pro').
:- use_module(library(porter_stem)).

%% dispatch event predicate
dispatch_event(Q,E) :-
    member('STOPPED',E),
    q_turn(Q,-30).

dispatch_event(Q,E) :-
    member('rays',E),
    nth1(9,E,X), % unify the 9th element of E with X, start counting at 1
    nth1(10,E,Y), % unify the 10th element of E with Y
    Dist is sqrt(abs(X)**2 + abs(Y)**2),
    write('ray> '),write(' distance '),write(Dist),nl,
    Dist =< 500,
    q_turn(Q,-30).

dispatch_event(_,__). % return true if we are not interested in the event

%% handle events predicate
handle_events(_,[]).

handle_events(Q,[H|T]) :-
    tokenize_atom(H,TokenList), % tokenize the string H
    dispatch_event(Q,TokenList),
    handle_events(Q,T).
Event Programming in Prolog

Given the token list of the rays event:

\[ \text{OK, (, ask, rays, 2, ), 1, worldspawn, 0, -209.469, 0, 2, worldspawn, -9.15527e-005, 286.469, 0} \]

We want to extract the list of objects with their corresponding data.

% parse the rays command event token list, ignoring % the first 6 tokens
q_parse_rays([_, _, _, _, _, _|TokenList], ObjList) :-
    q_parse_tokenlist(TokenList, ObjList).

% parse the object token list and construct a list % of object together with their corresponding data
q_parse_tokenlist([], []).
q_parse_tokenlist([I, O, X, Y, Z|Next], [Object|Tail]) :-
    Object = [I, O, X, Y, Z],
    q_parse_tokenlist(Next, Tail).

Given the rays command event above, what does the constructed ObjList look like?
Asynchronous Quagent Programming in Prolog
Asynchronous Prolog

Idea: let’s write a program that walks a quagent in a 100x100 square. We want to make sure that it reaches each of the corners.

```
run :-
    q_connect(Q),
    q_walk(Q,100),
    wait_until_stopped(Q),
    q_turn(Q,-90),
    q_walk(Q,100),
    wait_until_stopped(Q),
    q_turn(Q,-90),
    q_walk(Q,100),
    wait_until_stopped(Q),
    q_turn(Q,-90),
    q_walk(Q,100),
    wait_until_stopped(Q),
    q_turn(Q,-90),
    q_walk(Q,100),
    wait_until_stopped(Q),
    q_turn(Q,-90),
    q_walk(Q,100),
    wait_until_stopped(Q),
    q_turn(Q,-90),
    q_walk(Q,100),
    q_close(Q).
```

% keep polling the events
% base case
wait_until_stopped(Q) :-
    % could be doing something smart here
    q_events(Q,E),
    parse_stopped_event(E).
% recursive step
wait_until_stopped(Q) :-
    % could be doing something smart here
    wait_until_stopped(Q).

% parse event list for stopped event
% base cases
parse_stopped_event([]) :- fail.
parse_stopped_event([H|_]) :-
    tokenize_atom(H,TokenList),
    member('STOPPED',TokenList).
% recursive step
parse_stopped_event([_|T]) :-
    parse_stopped_event(T).
Idea: let’s write a program that walks a quagent in a 100x100 square. We want to make sure that it reaches each of the corners but now we use a FSM to control the movements.

This is a Moore machine: actions on the states, conditions on the transitions.
%%% define the machine
% initial and final states
initial(start).
final(stop).

% define the actions at the states
action(start,Q) :- writeln('start state'), q_walk(Q,100).
action(one,Q) :- writeln('state one'), q_turn(Q,-90), q_walk(Q,100).
action(two,Q) :- writeln('state two'), q_turn(Q,-90), q_walk(Q,100).
action(three,Q) :- writeln('state three'), q_turn(Q,-90), q_walk(Q,100).
action(stop,_):- writeln('stop state: all done').

% define the edges
edge(start,wait_stopped,one).
edge(one,wait_stopped,two).
edge(two,wait_stopped,three).
edge(three,wait_stopped,stop).

% define the condition for the transitions
condition(wait_stopped,Q) :- wait_until_stopped(Q).

%%% main loop which uses the finite state machine
state_machine(State,Q) :-
    final(State),
    action(State,Q).

state_machine(State,Q) :-
    action(State,Q),
    edge(State,Label,NextState),
    condition(Label,Q),
    state_machine(NextState,Q).

%%% run the program
run :-
    q_connect(Q),
    state_machine(start,Q),
    q_close(Q).

Full implementation available on the course website.
Preview of Coming Attractions

- Form Teams of 2-4
- Team Assignments
  - Prolog assignment ‘Tofu Deathmatch’
  - Pick an article to present in ‘Game Programming Wisdom 4’ – each team will have to present one article -- each team should pick 3 articles and then we figure out which team presents what.
  - You should also start thinking about Projects
    - example projects: [http://www.cs.rochester.edu/~brown/242/assts/assts.html](http://www.cs.rochester.edu/~brown/242/assts/assts.html)
  - Deathmatch competition - Thursday 3/18
For the ‘Tofu Deathmatch’ assignment, add the following lines to your ‘quagent.config’ file (same folder as your empty and obstacle rooms):

```
tofu 288 288 128
tofu 384 64 128
tofu 256 128 128
tofu 192 448 128
```