General Game Playing

Game Playing

Logic

Possible to use logical reasoning for game play
- Computing legality of moves
- Computing consequences of actions
- Computing goal achievement
- Computing termination

Easy to convert from logic to other representations
- many orders or magnitude speedup on simulation
- no asymptotic change

Having a formal description of a game is one thing; being able to use that description to play the game effectively is something else entirely. In this lesson, we discuss strategies for building general games players and some of the difficulties that need to be handled.

Since game descriptions are written in logic, it is obviously necessary for a game player to do some amount of automated reasoning. There are two extremes here. (1) One possibility is for the game player to process the game description interpretively throughout a game. (2) The second possibility is for the player to use the description to devise a specialized program and then use that program to play the game. This is effectively automatic programming. In this lesson, we discuss the first possibility and leave it to you to think about the second possibility and various hybrid approaches. We discuss the alternative approach later in the course.

Programme

Lesson 4 (this lesson):
- Infrastructure
- Creating a Legal Player
- Creating a Random Player

Lessons 5-8:
- Complete and Incomplete Search

Lessons 9-12
- Propositional Nets rather than State Machines

Lessons 13-16
- Logic rather than State Machines or Propnets

We begin the lesson by talking about some infrastructure that frames the problem more precisely. We then consider a couple of search-free uses of this infrastructure, viz. legal players and random players. In Lessons 5-8, we look at complete search techniques (which are appropriate for small game graphs) as well as incomplete search techniques (which are necessary for large game graphs). In Lessons 9-12, we examine some game playing techniques based on properties of states rather than monolithic states. Finally, in Lessons 13-16, we show ways that game descriptions can be used to deduce general properties of games without enumerating states or properties of those states.

Architecture

Listen Loop
- Receives messages from Game Manager
- Calls appropriate event handler
- Sends result back to Game Manager

Event handlers
- each handles a different type of message
- responds with time bounds

A game player is typically implemented as a web service. As soon as it begins running, the player enters a loop listening for messages from the Game Manager. Upon receipt of a message, the player calls an appropriate handler for that type of message. When the handler is finished, the player sends the return value to the Game Manager. Given this architecture, building a player means writing event handlers for the different types of messages in the GGP communication protocol.

Event Types

(info)

(start *id role ruleset startclock playclock*)

(play *id move*)

(stop *id move*)

(abort *id*)

There is typically one handler for each type of message in the GGP protocol - info, start, play, stop, and abort. So, that means writing 5 event handlers.

Predefined Subroutines

```
findroles(rules)
findbases(rules)
findinputs(role, rules)
findinits(rules)
findlegalp(role, action, state, rules)
findlegalx(role, state, rules)
findlegals(role, state, rules)
findnexts(move, state, rules)
findreward(role, state, rules)
findterminalp(state, rules)


doesify(roles, actions)
undoesify(sentence)
```

In order to facilitate the implementation of these message handlers, the GGP code base available in the course website contains definitions for the subroutines shown here. There are subroutines for computing the key components of a game. For example, findroles produces a list of roles in the game. findinits gives a list of sentences true in the initial state. There are also come utility subroutines to make our job easier. For example, doesify takes a list of roles and a list of actions as argument and produces a list of *sentences* stating that that each role specified in the roles argument executes the corresponding action in the actions argument. Undoesify reverses the process.

Event Handlers

```
function info()
  {...code that calls predefined subroutines...}

function start()
  {...code that calls predefined subroutines...}

function play()
  {...code that calls predefined subroutines...}

function stop()
  {...code that calls predefined subroutines...}

function abort()
  {...code that calls predefined subroutines...}
```

Your job in building a player is to use the subroutines provided to write the handlers called by the message listener. In the next two segments, we look at a couple of simple approaches for doing this. We will present the definitions in Javascript. You can use these yourself if you are building your player in Javascript or you can adjust as appropriate to implement the Java-based versions. Alternatively, you can click the appropriate box in the parametric player to select from among the different approaches. In this latter case, you do not need to type in any code, but you should look at the code so that you know what is going on when you make your selections.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Creating a Legal Player

Legal Player

Simple behavior

Maintains current state of the game.

On each step, selects first legal action it finds.

NB: Selects same action every time in a state.

A legal player is one of the simplest types of game player. In each state, a legal player selects an action based solely on its legality, without consideration of the consequences. Typically, the choice of action is consistent - it selects the same action every time it finds itself in the same state. (In this way, a legal player differs from a random player, which selects different legal actions on different occasions.)

Implementation

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!='nil') {state=findnexts(move,state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

Using the basic subroutines provided in the GGP starter pack, building a legal player is simple. Here we see the entire implementation. Throughout this course, we use Javascript for Computer code. This is a compromise between Lisp and Java. Lisp is a powerful language, is highly efficient and is easy to use in building players; however, it is not very widely known. Building programs in Java is more difficult, but the language is familiar to more people these days. Javascript has the flexibility of Lisp and at the same time is easily converted to Java. It can also be used directly to implement players in Web browsers or in standalone applications. Let's walk through this implementation.

Initialization

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

We start by setting up some global variables to maintain state while a match is in progress. There is a variable to hold the game rule set, a variable to hold the player's role in that game, and a variable to hold the current state. (Properly, we should create a data structure for each match; and we should attach these values to this data structure. However, we are striving for simplicity of implementation in our presentation here.)

Info

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

Next, we define a handler for each type of message. The info handler does nothing and simply returns ‘ready’.

Start

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,move)
 {if (move!='nil')
     {state=findnexts(doesify(roles,move),state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

The start event handler assigns values to the ruleset and role variables based on the incoming start message; it initializes roles and the state variable; and it then returns ready, as required by the GGP protocol.

Play

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

The play event handler takes a match identifier and a move as arguments. If the move is not ‘nil’, and it uses nexts to compute the state resulting from the preceding state and the actions supplied in the move. Once the player has the latest state, it uses legalx to compute a legal move.

Stop

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

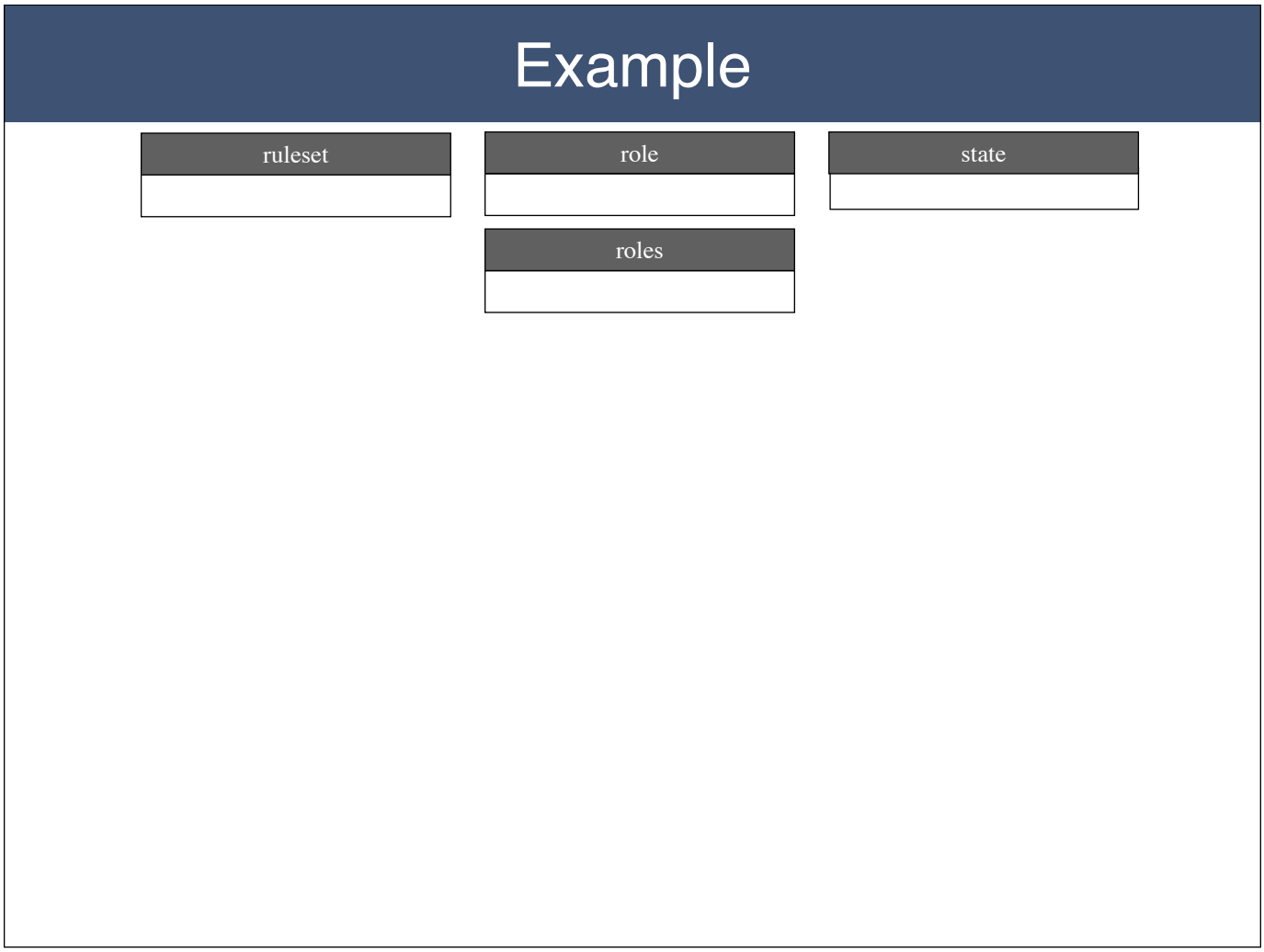
The stop event handler for our legal player does nothing. It ignores the inputs and simply returns done as required by the GGP protocol.

Abort

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

Like the stop message handler, the abort message handler for our player also does nothing. It simply returns done.

Example



Just to be clear on how this works, let's work through a short Tic-Tac-Toe match. When a player is initialized, it sets up data structures to hold the game description, the role, and the state. These are initially empty. The player does not retain startclock and playclock since it does not do extensive computation for which those limits are relevant.

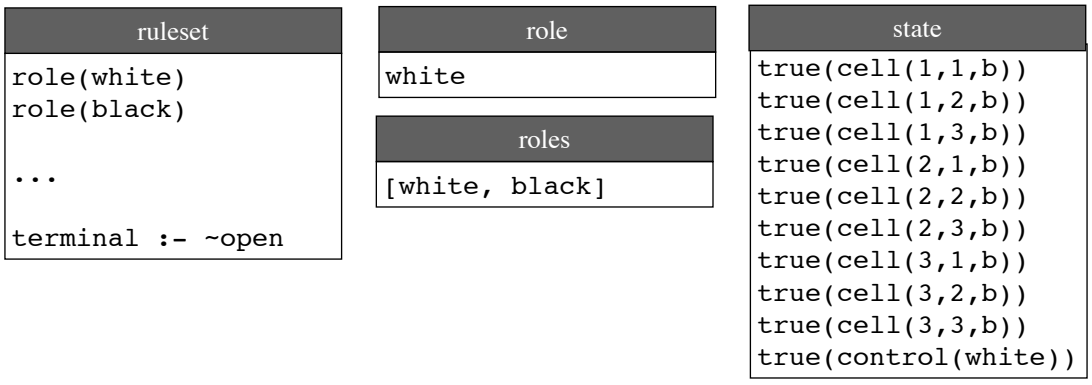
Example

Message: `start(m23, white, [role(white),role(black),...], 10, 10)`

Let's assume that the player receives a start message from the Game Manager of the sort shown below. The match identifier is m23. The player is told to be the white player. There are the usual axioms of Tic-Tac-Toe. The startclock and playclock are both 10 seconds.

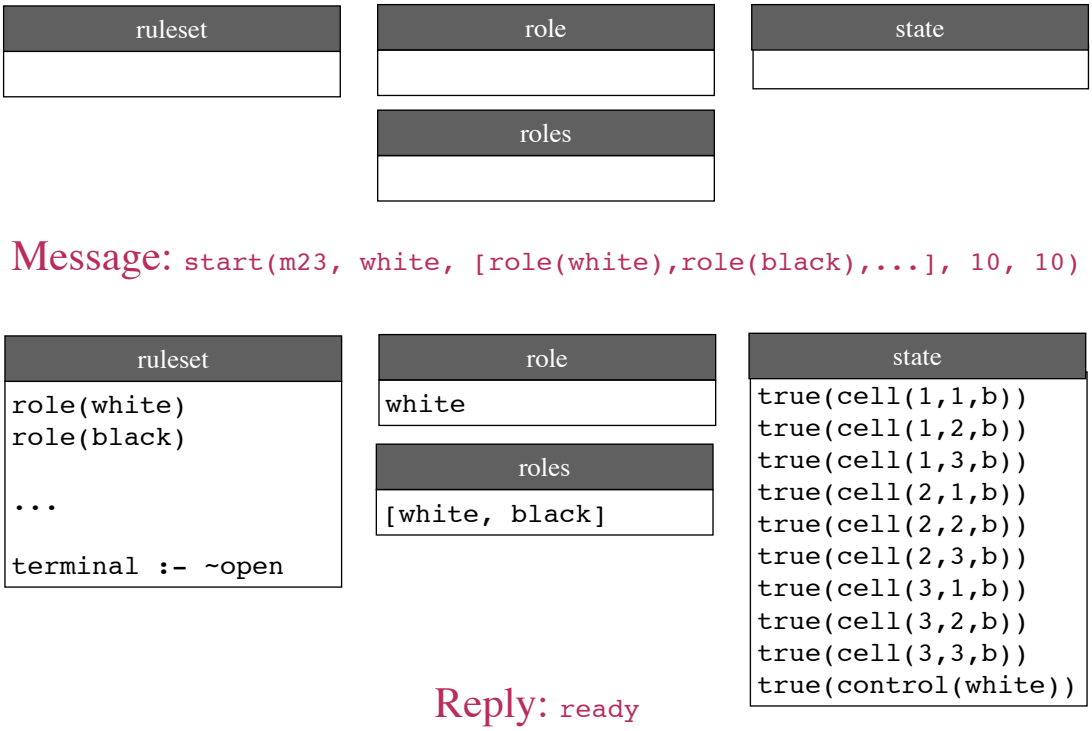
Example

Message: `start(m23, white, [role(white),role(black),...], 10, 10)`



On receipt of this message, the player code calls the start handler. This records the game description and the player's role in the corresponding global variables. It also computes and saves the initial state.

Example



The returned value is ready, which is then sent back to the Game Manager.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Play begins after all of the players have responded or after the startclock has expired, whichever comes first.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

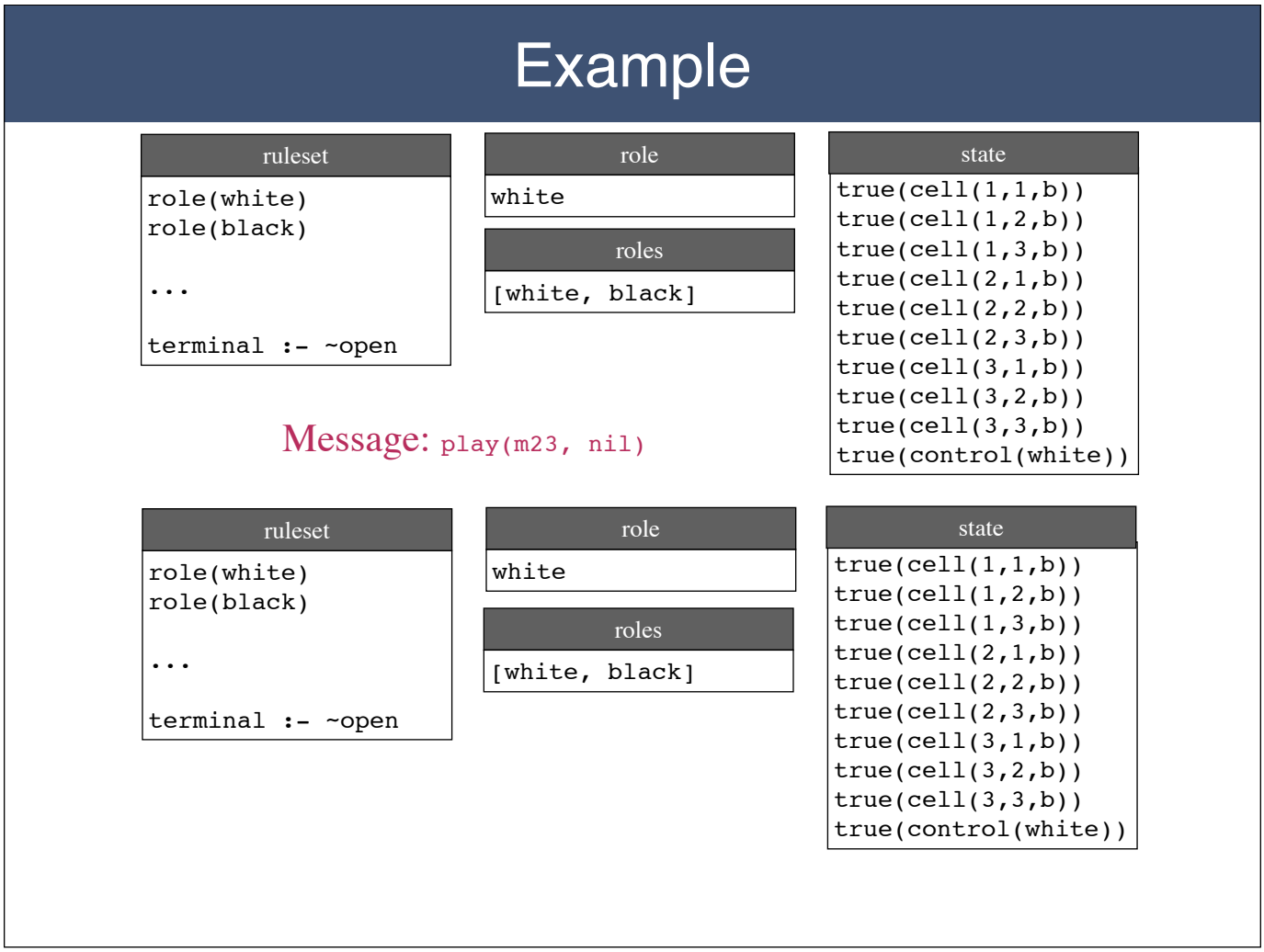
state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Message: `play(m23, nil)`

Once the Game Manager is ready, it sends a suitable play message to all players. Here we have a request for the player to choose an action for match m23. The value nil signifies that this is the first step of the match.

Example



On receipt of this message, the player code invokes the play handler with the arguments passed to it by the Game Manager. Since the move argument is nil, no changes are made to the data structures.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Message: `play(m23, nil)`

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Reply: `mark(1,1)`

Using this state, together with the role and description associated with this match, the player then computes the first legal move, i.e. mark(1,1) and returns that as answer.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

The Game manager checks that the actions of all players are legal, simulates their effects and updates the state of the game, and then sends play messages to the players to solicit their next actions.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Message: `play(m23, [mark(1,1),noop])`

In this case, the player will receive the message shown here.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Message: `play(m23, [mark(1,1),noop])`

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,x))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(black))
```

Again, the player invokes its play handler with the arguments passed to it by the Game Manager. This time, the move is not nil, and so the player uses findnexts to compute the next state. This results in the dataset shown here on the lower right.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Message: `play(m23, [mark(1,1),noop])`

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,x))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(black))
```

Reply: `noop`

Using this state, the player then computes the first legal action in this state, its only legal action, viz. noop, and returns that as answer.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,x))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(black))
```

This process then repeats until the end of the game.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,x))
true(cell(1,2,o))
true(cell(1,3,o))
true(cell(2,1,b))
true(cell(2,2,x))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Message: `stop(m23, [mark(3,3),noop])`

When the game is over, the player receives a message like the one shown here.

Example

ruleset

```
role(white)
role(black)

...

terminal :- ~open
```

role

```
white
```

roles

```
[white, black]
```

state

```
true(cell(1,1,x))
true(cell(1,2,o))
true(cell(1,3,o))
true(cell(2,1,b))
true(cell(2,2,x))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Message: `stop(m23, [mark(3,3),noop])`

Reply: `done`

While some players are able to make use of the information in a stop message, our legal player simply ignores this information and returns done, terminating its activity on this match.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Creating a Random Player

Random Player

Simple behavior

- Maintains current state of the game.
- On each step, selects random legal action it finds.

NB: May take different action each time in a state.

A random player is similar to a legal player in that it maintains a state and selects an action for each state based solely on its legality, without consideration of the consequences. A random player differs from a legal player in that it does not simply take the first legal move it finds but rather selects randomly from among the legal actions available in the state, usually choosing a different move on different occasions.

Implementation

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  var actions=findlegals(role,state,game);
  return actions[randomindex(actions.length)]}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

The implementation of a random player is almost identical to the implementation of a legal player. The only difference is in the play handler.

Play Handler

Legal Player:

```
function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!='nil') {state=findnexts(move,state,ruleset)};
  return undoesify(findlegalx(role,state,ruleset))}
```

Random Player:

```
function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!='nil') {state=findnexts(move,state,ruleset)};
  var actions=findlegals(role,state,game);
  return actions[randomindex(actions.length)]}
```

In a legal player, the play handler simply returns the first legal move. In a random player, the play handler first computes all legal moves and then selects one at random.

Remarks on Random Players

Random Players are no "smarter" than legal players.
- Appear more interesting because unpredictable.
- Sometimes avoid traps that befall consistent players like Legal.
- Often used as a comparison to show that a player or method performs statistically better than chance.

Random players take slightly more time than legal
- Compute all legal actions rather than just one
- Noticeable only on games with many legal actions

Random players are no "smarter" than legal players. However, they often appear more interesting because they are unpredictable. Also, they sometimes avoid traps that befall consistent players like legal, which can sometimes maneuver themselves into corners from which they are unable to escape. Random players are also used as standards to show that general game players or specific methods perform better than chance.

A random player consumes slightly more compute time than a legal player, since it must compute all legal moves rather than just one. For most games, this is not a problem; but for games with a large number of possible actions, the difference can be noticeable.


GENERAL
GAME
PLAYING