General Game Playing

Game Description

General Game Playing

General Game Players are systems able to play arbitrary games effectively based solely on formal descriptions supplied at “runtime”.

Translation: They don’t know the rules until the game starts.

The defining characteristic of General Game Playing is that players do not know the rules of games before those games begin. Game rules are communicated at "runtime", and the players must be able to read and understand the descriptions they are given in order to play legally and effectively. This characteristic carries with it the requirement of language for writing game rules.

Game Description Language

Game Description Language (or GDL) is a formal language for encoding the rules of games.

GDL is widely used in the research literature and is used in virtually all General Game Playing competitions.

GDL extensions are applicable in real-world applications such as Enterprise Management and Computational Law.

In general game playing, information about games is typically communicated to players in a formal language called Game Description Language, or GDL. As mentioned in the Introduction, GDL is widely used in the research literature and is used in virtually all General Game Playing competitions. Moreover, it forms the basis for some more expressive variants that have significant value in real-world applications, such as Enterprise Management and Computational Law.

GDL is a Logic Programming Language

Tic-Tac-Toe

```
role(x)
role(o)

init(cell(1,1,b))
init(cell(1,2,b))
init(cell(1,3,b))
init(cell(2,1,b))
init(cell(2,2,b))
init(cell(2,3,b))
init(cell(3,1,b))
init(cell(3,2,b))
init(cell(3,3,b))
init(control(x))

legal(P,mark(X,Y)) :-
  true(cell(X,Y,b)) &
  true(control(P))

legal(x,noop) :-
  true(control(o))

legal(o,noop) :-
  true(control(x))
```

```
next(cell(M,N,P)) :-
    does(P,mark(M,N))

next(cell(M,N,Z)) :-
    does(P,mark(M,N)) &
    true(cell(M,N,Z)) & Z!=b

next(cell(M,N,b)) :-
    does(P,mark(J,K)) &
    true(cell(M,N,b)) &
    distinct(M,J)

next(cell(M,N,b)) :-
    does(P,mark(J,K)) &
    true(cell(M,N,b)) &
    distinct(N,K)

next(control(x)) :-
    true(control(o))

next(control(o)) :-
    true(control(x))

terminal :- line(P)
terminal :- ~open

goal(x,100) :- line(x) & ~line(o)
goal(x,50) :- ~line(x) & ~line(o)
goal(x,0) :- ~line(x) & line(o)
goal(o,100) :- ~line(x) & line(o)
goal(o,50) :- ~line(x) & ~line(o)
goal(o,0) :- line(x) & ~line(o)
```

```
row(M,P) :-
    true(cell(M,1,P)) &
    true(cell(M,2,P)) &
    true(cell(M,3,P))

column(N,P) :-
    true(cell(1,N,P)) &
    true(cell(2,N,P)) &
    true(cell(3,N,P))

diagonal(P) :-
    true(cell(1,1,P)) &
    true(cell(2,2,P)) &
    true(cell(3,3,P))

diagonal(P) :-
    true(cell(1,3,P)) &
    true(cell(2,2,P)) &
    true(cell(3,1,P))

line(P) :- row(M,P)
line(P) :- column(N,P)
line(P) :- diagonal(P)

open :-
true(cell(M,N,b))
```

As we shall see, GDL is a logic programming language. Game descriptions are logic programs consisting of rules that define the key elements of games, such as initial conditions, move legality, game dynamics, rewards, and termination.

Programme

Logic Programs in GDL

Game Description Example
Game Simulation Example

Game Requirements
Prefix GDL

This lesson is an overview of GDL. We begin with a detailed introduction to GDL. We then look at a sample game description, and we look at the use of this description in simulating a match of the game. Finally, we talk about additional requirements on games that ensure that they are interesting; and we summarize the prefix syntax for GDL used in most GGP competitions.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Game Description Language

Logic Programs

A *logic program* is a collection of logical *rules* that define relations either (1) directly by enumeration or (2) indirectly in the form of rules.

A logic program is a collection of logical rules that define relations either (1) directly by enumeration or (2) indirectly in the form of rules.

Simple Example

Data:

```
parent(art,bob)
parent(amy,bob)
parent(bob,cal)
parent(bob,coe)
```

Rule:

```
grandparent(X,Z) :- parent(X,Y) & parent(Y,Z)
```

Output:

```
grandparent(art,cal)
grandparent(art,coe)
grandparent(amy,cal)
grandparent(amy,coe)
```

Here is a simple example. The first four sentences here define the parent relation by enumeration. Art is the parent of Bob, and Amy is the parent of Bob. Bob is the parent of Cal and Coe. The rule in the middle defines the grandparent relation in terms of the parent relation. X is the grandparent of Z if X is a parent of some person Y and Y is a parent of Z. As we shall see, given these definitions of parent and grandparent, we can conclude that Art and Amy are the grandparents of Cal and Coe.

GDL Similar to Prolog but ...

Semantics is purely declarative
- No assert or retract
- No cut

Restrictions assure decidability of pertinent questions
- Safety
- Stratified Recursion
- Stratified Negation
- No nested functional terms

Reserved Words
- role, init, legal, next, goal, terminal, ...
- described in next segment

GDL is a logic programming language and in many ways resembles Prolog, the most popular logic programming language. However, there are several important differences. (1) First of all, the semantics of GDL is purely declarative, i.e. there are no imperative features, such as assert, retract, and cut. (2) Second, GDL has restrictions that ensure that all pertinent questions are decidable; in particular, all relations can be computed in finite time. (3) Finally there are some reserved words, which tailor the language to the task of defining games. Despite these restrictions, we frequently use the phrase logic program to refer to a game descriotion in GDL.

Vocabulary

Components:

- Object Constants: `a`, `b`, `c`
- Function Constants: `f`, `g`, `h`
- Relation Constants: `p, q, r`
- Variables: `X`, `Y`, `Z`

The *arity* of a function constant or relation constant is the number of arguments that can be associated with the function constant or relation constant in writing complex expressions in the language.

Logic programs in GDL are built up from four disjoint classes of symbols, viz. object constants, function constants, relation constants, and variables. In what follows, we write such symbols as strings of letters, digits, and a few non-alphanumeric characters (e.g. "_"). Constants must begin with a lower case letter or digit. Variables must begin with an upper case letter.

The arity of a function constant or relation constant is the number of “arguments” that can be “associated” with the function or relation when writing expressions (as we shall see).

Syntax

Terms:

- Object Constants: `a`, `b`
- Variables: `X`, `Y`, `Z`
- Functional Terms: `f(a)`, `g(X,b)`

In traditional logic programs, functional terms can be nested within other functional terms, but this is not permitted in GDL.

A term is either an object constant, a variable, or a functional term. A functional term is an expression consisting of an n-ary function constant and n simpler terms. In what follows, we write functional terms in traditional mathematical notation - the function constant followed by its arguments enclosed in parentheses and separated by commas. For example, if f is a unary function constant and g is a binary function constant, if a and b are object constants, and if X is a variable, then f(a) is a functional term, and so is g(X,b).

Atoms, Negations, and Literals

Atoms

```
p(a,b), p(a,f(a)), p(g(a,b),c)
```

Negations

```
~p(a,b)
```

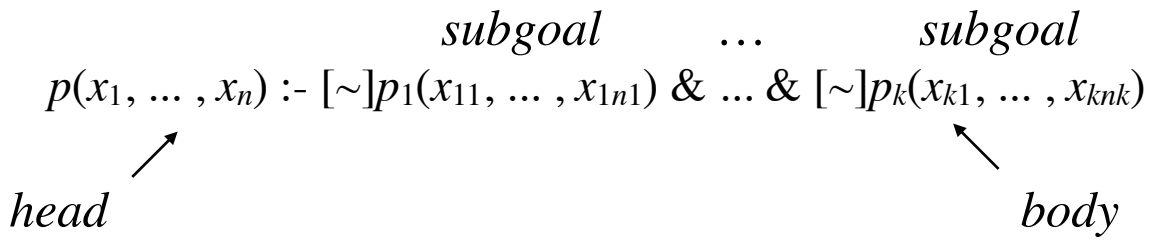
A *literal* is either an atom or a negation of an atom.

```
p(a,b), ~p(a,b)
```

An atom is a *positive literal*.
A negations is a *negative literal*.

An atom is an expression consisting of an n-ary relation constant and n terms. In what follows, we write atoms in traditional mathematical notation - the relation constant followed by its arguments enclosed in parentheses and separated by commas. For example, if p is a binary relation constant and if a and b are object constants, then p(a,b) is an atom. A negation is an expression formed using the negation sign ~ and an atom. For example, ~p(a,b). A literal is either an atom or a negation. An atom is sometimes called a positive literal, and a negation is sometimes called a negative literal.

Rules



A rule is an expression consisting of a distinguished atom, called the head, and a conjunction of zero or more literals, called the body, separated by the :- operator. The literals in the body are called subgoals. The intended meaning is that an instance of the head is true whenever corresponding instances of all of the positive subgoals are true and all of the negative subgoals are false.

GDL Descriptions

A *logic program* is a finite set of atoms and rules (subject to conditions described shortly).

Example:

```
p(X,Y) :- f(X,Y)
p(X,Y) :- m(X,Y)
```

Example:

```
g(X,Z) :- p(X,Y) & p(Y,Z)
```

Example:

```
a(X,Z) :- p(X,Z)
a(X,Z) :- p(X,Y) & a(Y,Z)
```

Example:

```
ra(X,Y) :- a(X,Y) & ~p(X,Y)
```

A logic program in GDL is a finite set of atoms and rules of this form. In order to simplify our definitions and analysis, we occasionally talk about infinite sets of rules. While these sets are useful, they are not themselves logic programs. Here are some examples. In this first example, we define the parent relation p in terms of father f and mother m. In the second example, we define grandparent in terms of parent. In the third, we define ancestor in terms of parent. Note that the definition in this case is recursive. Finally, we define remote ancestor as any ancestor that is not a parent.

Although every GDL description is a finite set of atoms and rules, not every finite set of atoms and rules is a GDL description. As mentioned earlier, there are some restrictions to ensure that such descriptions have desirable properties.

Safety

A rule is *safe* if and only if every variable in the head appears in some positive subgoal in the body.

Safe Rule:

```
r(X,Z) :- p(X,Y) & q(Y,Z) & ~r(X,Y)
```

Unsafe Rule:

```
r(X,Z) :- p(X,Y) & q(Y,X)
```

Unsafe Rule:

```
r(X,Y) :- p(X,Y) & ~q(Y,Z)
```

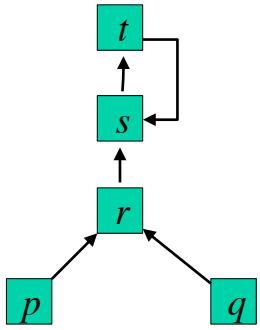
In GDL, we require all rules to be safe.

The first of these restrictions is called safety. A rule in a logic program is safe if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body. The first rule shown here is safe. Variables X and Z appear in the head and Y appears in a negative subgoal. Fortunately, all three of those variables appear in positive subgoals as well, and so the rule is safe. The second is not safe because variable Z appears in the head but not in any positive subgoal. The third rule is not safe because the variable Z appears in a negative subgoal but not in a positive subgoal. In GDL, we require all rules to be safe.

Dependency Graph

The *dependency graph* for a set of rules is a directed graph in which (1) the nodes are the relations mentioned in the head and bodies of the rules and (2) there is an arc from a node *p* to a node *q* whenever *p* occurs with the body of a rule in which *q* is in the head.

```
r(X,Y) :- p(X,Y) & q(X,Y)
s(X,Y) :- r(X,Y)
s(X,Z) :- r(X,Y) & t(Y,Z)
t(X,Z) :- s(X,Y) & s(Y,X)
```



A set of rules is *recursive* if it contains a cycle. Otherwise, it is *non-recursive*.

The next two restrictions on GDL descriptions have to do with recursion. The restrictions are best defined using the notion of dependency graphs. The dependency graph for a set of rules is a directed graph in which (1) the nodes are the relations mentioned in the head and bodies of the rules and (2) there is an arc from a node p to a node q whenever p occurs with the body of a rule in which q is in the head. A set of rules is recursive if and only if its dependency graph contains a cycle.

Stratified Recursion

The recursion in a set of rules is said to be *stratified* if and only if every variable in a subgoal relation occurs in a subgoal with a relation at a lower stratum.

Stratified Recursion:

```
r(X,Z) :- p(X,Y) & q(Z) & r(Y,Z)
```

Recursion that is not stratified:

```
r(X,Z) :- r(X,Y) & r(Y,Z)
```

In GDL, we require that all recursions be stratified.

A recursion in a set of rules is said to be stratified if and only if every variable in a subgoal relation occurs in a subgoal with a relation at a lower stratum. The recursion in the first rule shown here is stratified because all of the variables involved in the recursion occur in relations that are not defined in terms of r. The recursion in the second rule is not stratified because the variables involved in the recursion do not appear in other relations. In GDL, we require that all recursions be stratified.

Stratified Negation

The negation in a set of rules is said to be *stratified* if and only if there is no recursive cycle in the dependency graph involving a negation.

Negation that is not stratified:

```
r(X,Z) :- p(X,Y,Z)
r(X,Z) :- p(X,Y,Z) & ~r(Y,Z)
```

Stratified Negation:

```
t(X,Y) :- q(X,Y) & ~r(X,Y)
r(X,Z) :- p(X,Y)
r(X,Z) :- r(X,Y) & r(Y,Z)
```

In GDL, we require that all negations be stratified.

A negation in a set of rules is said to be stratified if and only if there is no recursive cycle in the dependency graph involving a negation. For example, the first rule set shown here is not stratified because there is a cycle involving a negative occurrence of r. By contrast, the second set of rules is stratified. The rule set is recursive, but there is no negation in the cycle. The only negative occurrence of r occurs in the definition of t and is not part of any recursion. In GDL, we require all negations to be stratified.

Herbrand Universe

The *Herbrand universe* for a logic program is the set of all ground terms in the language.

Example 1:

Object Constants: `a`, `b`

Herbrand Universe: `a`, `b`

Example 2:

Object Constants: `a`, `b`

Unary Function Constants: `f`

Binary Function Constant: `g`

Herbrand Universe: `a`, `b`, `f(a)`, `f(b)`, `g(a,a)`, ...

With these definitions behind us, we can formalize the semantics, i.e. the meaning, of GDL descriptions. We start with the notion of a Herbrand universe. The Herbrand universe is the set of all ground terms in the language. In the case of a language without function constants, the Herbrand universe is exactly the set of all object constants. In the presence of function constants, we add in the functional terms that can be formed using those function constants. Since nested functional terms are forbidden in GDL, the Herbrand universe is always finite.

Herbrand Base

The *Herbrand base* for a logic program is the set of all ground atoms in the language.

Object Constants: `a`, `b`
Unary Relation Constant: `p`
Binary Relation Constant: `q`
Herbrand Universe: `a`, `b`

Herbrand Base:

```
{p(a),p(b),q(a,a),q(a,b),q(b,a),q(b,b)}
```

The Herbrand base for a logic program is the set of all ground atoms that can be formed from the relation constants in the language and the elements of its Herbrand universe. For example, given the language shown here, the Herbrand base consists of the six atoms p(a), p(b), q(a,a), and so forth.

Herbrand Interpretations

An *interpretation* for a logic program is an arbitrary subset of the Herbrand base for the program.

Herbrand Base:

{p(a), p(b), q(a,a), q(a,b), q(b,a), q(b,b)}

Interpretation 1:

{p(a), q(a,b), q(b,a)}

Interpretation 2:

{p(a), p(b), q(a,a), q(a,b), q(b,a), q(b,b)}

Interpretation 3:

{}

A Herbrand interpretation is an arbitrary subset of the Herbrand base for a program. Intuitively, we can think of a Herbrand interpretation as listing the atoms that are true in that interpretation. Given the Herbrand base we just saw, here are three different interpretations. Since there are 6 atoms in the Herbrand base, there are 2^6 or 64 distinct interpretations.

Herbrand Models

An interpretation Δ *satisfies* a ground sentence (i.e. it is a model) under the following conditions:

Δ satisfies an atom φ iff $\varphi \in \Delta$.

Δ satisfies $\sim\varphi$ iff Δ does *not* satisfy φ.

Δ satisfies φ_1 & … & φ_n iff Δ satisfies *every* φ_i.

Δ satisfies φ_2 :- φ_1 iff

Δ satisfies φ_2 whenever it satisfies φ_1.

An interpretation *satisfies* a non-ground sentence if and only if it satisfies every ground instance of that sentence.

We say that an interpretation Delta satisfies an expression under the following conditions. Delta satisfies a ground atom phi if and only if phi is in Delta. Delta satisfies ~phi if and only if phi is *not* in Delta. Delta satisfies phi1 and phi2 and so forth if and only if it satisfies every phi_i. Finally, Delta satisfies a rule if and only if it satisfies the head or fails to satisfy the body. Equivalently, Delta satisfies a rule if and only if it satisfies the head whenever it satisfies the body. Finally, we say that Delta satisfies a rule with variables if and only if it satisfies very ground instance.

Multiple Models

In general, a logic program can have more than one model.

Logic Program:

```
{p(a,b), q(X,Y):-p(X,Y)}
```

Model 1: `{p(a,b), q(a,b)}`

Model 2: `{p(a,b), q(a,b), q(b,a)}`

In general, a logic program can have more than one model. Consider the program consisting of p(a,b) and the rule q(X,Y) if p(X,Y). This program has one model with just p(a,b) and q(a,b), and it has another model with p(a,b) and q(a,b) and q(b,a). And it has other models as well. However, it is worth noting that the first model is a subset of the second. It is clear that p(a) and q(a,b) must be in any model of this program, but q(b,a) is optional.

Minimality

To eliminate ambiguity, we adopt a minimal model semantics. A model D is a *minimal model* of a program O iff no proper subset of D is a model of O.

Logic Program:
`{p(a,b), q(X,Y):-p(X,Y)}`

Good Model 1: `{p(a,b), q(a,b)}`
Bad Model 2: `{p(a,b), q(a,b), q(b,a)}`

Theorem: Given our various restrictions, every logic program has a unique minimal model. Therefore, unambiguous answers. Also, all models are finite!

To eliminate such ambiguities, we usually adopt a minimal model semantics for logic programs. A model D of a program O is minimal if and only if no proper subset of D is a model of O. One interesting property of our language is that every logic program has a unique minimal model. Also all models are finite.

Open versus Closed Logic Programs

Logic programs as just defined are *closed* in that they fix the meaning of all relations.

In *open* logic programs, some of the relations are given as inputs (rather than being defined) and other relations are produced as outputs (defined in terms of the inputs). In other words, open logic programs can be used with different input datasets to produce different output datasets.

Logic programs as just defined are closed in that they fix the meaning of all relations in the program. In open logic programs, some of the relations (the inputs) are undefined, and other relations (the outputs) are defined in terms of these. The same program can be used with different input relations, yielding different output relations in each case.

Open Logic Programs

An open logic program is a logic program together with a partition of the relations into two types - *input relations* and *output relations*.

Output relations can appear anywhere in the program, while input relations can appear only in the bodies of rules (never in the heads of rules).

Formally, an open logic program is a logic program together with a partition of the relation constants into two types - input relations and output relations. Output relations can appear anywhere in the program, but input relations can appear only in the subgoals of rules, not in their heads. The input base for an open logic program is the set of all atoms that can be formed from the base relations of the program and the entities in the program's domain. An input model is an arbitrary subset of its input base. The output base for an open logic program is the set of all atoms that can be formed from the output relations of the program and the entities in the program's domain. An output model is an arbitrary subset of its output base.

Inputs and Outputs

An *overall model* of an open logic program O with an input model D is is the minimal model of O∪D.

The *output* of an open logic program for input D is the subset of the overall model of O and D that mentions only output relations.

Given an open logic program P and an input model D, we define the overall model corresponding to D to be the minimal model of P U D. The output model corresponding to D is the intersection of the overall model with the program's output base; in other words, it consists of those sentences in the overall model that mention only the output relations.

Example

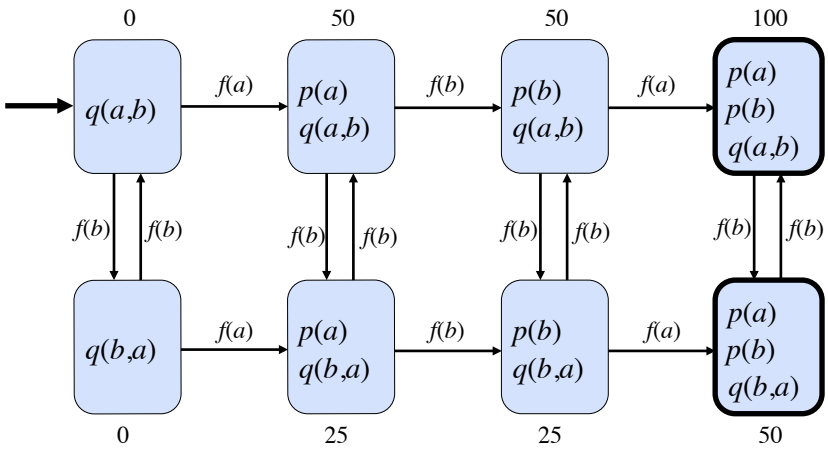
Base Relations: `p, q`
View Relations: `r`
Rules: `{r(X,Y) :- p(X,Y) & ~q(Y)}`

Input 1: `{p(a,b),p(b,b),q(b)}`
Output 1: `{r(a,b)}`

Input 2: `{p(a,b),p(b,b)}`
Output 2: `{r(a,b),r(b,b)}`

Finally, we can think of the meaning of an open logic program as a function that maps each input model for the program into the corresponding output model. For example, the simple logic program shown here gives different outputs for different inputs. Given p(a,b) and p(b,b) and q(b), the output is r(a,b). Given just p(a,b) and p(b,b), the output is r(a,b) and r(b,b).

Structured State Machine



Okay, now back to game description. As mentioned in the introduction, we can conceptualize a game as a structured state graph, like the one shown here. GDL gives us a way of describing such graphs in compact form.

Game-Independent Vocabulary

Relation Constants:

```
role(role)
base(proposition)
input(role,action)
init(proposition)
true(proposition)
next(proposition)
legal(role,action)
does(role,action)
goal(role,number)
terminal
```

Object Constants:

```
0, 1, 2, 3, … , 100 - numbers (i.e. entities)
```

The content of a structured state graph can be expressed in GDL using some reserved, game-independent vocabulary. The vocabulary is game-independent in that the same words are used in describing all games. There are ten game-independent relation constants, viz. the ones shown here. For example, role(a) means that a is a role in the game. base(p) means that p is a proposition in the game. action(a) means that a is an action in the game. init(p) means that the proposition p is true in the initial state. true(p) means that the proposition p is true in the current state. does(r,a) means that player r performs action a in the current state. next(p) means that the proposition p is true in the next state. legal(r,a) means it is legal for role r to play action a in the current state. goal(r,n) means that player the current state has utility n for player r. terminal means that the current state is a terminal state. GDL has no game-independent function constants. However, there are 101 game-independent object constants in GDL, viz. the base ten representations of the integers from 0 to 100, inclusive, i.e. 0, 1, 2, ... , 100. These are included for use as utility values for game states, with 0 being low and 100 being high.

Game-Specific Vocabulary

Object constants:

Roles: `white, black`

Entities in propositions and actions: `a, b, c`

Function Constants:

Relationships in propositions: `p, q`

Operations in actions: `f, g`

Relation Constants:

Helper relations: `r, s`

The first step in writing a game description is to select some game specific vocabulary to capture the structure of states and actions. For example, here we have object constants white and black as names for the two roles of our game. We use a, b, and c to refer to primitive entities in the game. We have names p and q for relationships among primitive entities and we have f and g as operations that can be performed on primitive entities. Finally, we have some helper relations r and s.

Propositions and Actions

Propositions and actions are terms in GDL.

Object constants: `a, b, c`
Relationship functions: `p, q`
Action functions: `f, g`

Propositions as terms: `p(a), q(b,c)`
Actions as terms: `f(a), g(b,c)`

Logic program rules say which propositions are true and which actions are performed (as shown in what follows) and specify the results of performing actions in states.

The propositions comprising states can be thought of as either object constants or functional terms in our language formed using relationship functions. Similarly, actions can be either object constants or functional terms formed using action functions. For example, applying relationship functions p and q to primitive objects a, b, and c, we end up with propositions like p(a) and q(b,c). And applying operations f and g to these objects, we end up with actions f(a) and g(b,c) and so forth. The rules in a game description say which propositions are true and which actions are performed (as shown in what follows) and specify the results of performing actions.

GDL Descriptions

A *game description* in GDL is an open logic program using GDL's game-independent and the game's game-specific vocabulary subject to the following constraints.

Inputs: `true`, `does`
Outputs: `role`, `base`, `input`, `init`, `legal`, `next`, `goal`, `terminal`

(1) `role`, `base`, `input`, `init` - independent of `true` and `does`
(2) `legal`, `goal`, `terminal` - depend on `true`
(3) `next` - depends on `true` and `does`

Finally, we define a game description as an open logic program using GDL's game-independent vocabulary together with the game's game-specific vocabulary subject to the following restrictions. true and does are the input relations to the program. (1) A GDL game description must give complete definitions for role, action, base, init. (2) It must define legal and goal and terminal in terms of an input true relation. (3) It must define next in terms of input true and does relations. Since does and true are treated as inputs, there must not be any rules with either of these relations in the head.

Okay, that's a lot to digest. In the abstract, these concepts are difficult to master, and they are not well-motivated. In practice, things are much simpler. In the next couple of segments, we illustrate these notions by looking at a specific game description and how it is used in simulating a game.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Game Description Example

Game-Specific Vocabulary

Object constants:

```
white, black - roles
1, 2, 3 - indices of rows and columns
x, o, b - marks
noop - no-op action
```

Function Constants:

```
cell(index,index,mark) --> proposition
control(role) --> proposition
mark(index,index) --> action
```

Relation Constants:

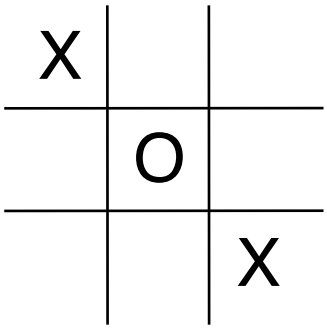
```
row(index,mark)
column(index,mark)
diagonal(mark)
line(mark)
open
```

Let's look at GDL in the context of a specific game, viz. Tic-Tac-Toe. As fundamental entities, we include white and black (the roles of the game), 1, 2, 3 (indices of rows and columns on the Tic-Tac-Toe board), and x, o, b (meaning blank).

We use the ternary function constant cell together with a row index and a column index and a mark to designate the proposition that the cell in the specified row and column contains the specified mark. For example, the datum cell(2,3,o) asserts that there is an o in the cell in row 2 and column 3. We use control to say whose turn it is to mark a cell. For example, the proposition control(white) asserts that it is white's turn.

In Tic-Tac-Toe, there only two types of actions a player can perform - it can mark a cell or it can do nothing (which is what a player does when it is not his turn to mark a cell). The binary relation mark together with a row m and a column n designates the action of placing a mark in row m and column n. The mark placed there depends on who does the action. The object constant noop refers to the act of doing nothing. Finally, we have some helper vocabulary, whose purpose will soon become clear.

State Representation




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

A state of a game is an arbitrary subset of the system's propositions. The propositions in a state are assumed to be true whenever the game in is that state, and all others are assumed to be false. For example, we can describe the Tic-Tac-Toe state shown here on the left with the set of propositions shown on the right.

Rules of Tic-Tac-Toe

```
role(white)
role(black)

base(cell(M,N,Z)) :-
  index(M) &
  index(N) &
  filler(Z)
base(control(W)) :- role(W)

input(W,mark(X,Y)) :-
  role(W) &
  index(X) &
  index(Y)

input(W,noop) :- role(W)

init(cell(X,Y,b)) :-
  index(X) &
  index(Y)

init(control(white))

legal(P,mark(X,Y)) :-
  true(cell(X,Y,b)) &
  true(control(P))

legal(x,noop) :-
  true(control(black))

legal(o,noop) :-
  true(control(white))
```

```
next(cell(M,N,x)) :-
    does(white,mark(M,N))

next(cell(M,N,0)) :-
    does(black,mark(M,N))

next(cell(M,N,Z)) :-
    does(P,mark(M,N)) &
    true(cell(M,N,Z)) & Z!=b

next(cell(M,N,b)) :-
    does(P,mark(J,K)) &
    true(cell(M,N,b)) &
    distinct(M,J)

next(cell(M,N,b)) :-
    does(P,mark(J,K)) &
    true(cell(M,N,b)) &
    distinct(N,K)

next(control(white)) :-
    true(control(black))

next(control(black)) :-
    true(control(white))

goal(white,100) :- line(x) & ~line(o)
goal(white,50) :- ~line(x) & ~line(o)
goal(white,0) :- ~line(x) & line(o)
goal(black,100) :- ~line(x) & line(o)
goal(black,50) :- ~line(x) & ~line(o)
goal(black,0) :- line(x) & ~line(o)
```

```
terminal :- line(P)
terminal :- ~open

row(M,P) :-
    true(cell(M,1,P)) &
    true(cell(M,2,P)) &
    true(cell(M,3,P))

column(N,P) :-
    true(cell(1,N,P)) &
    true(cell(2,N,P)) &
    true(cell(3,N,P))

diagonal(P) :-
    true(cell(1,1,P)) &
    true(cell(2,2,P)) &
    true(cell(3,3,P))

diagonal(P) :-
    true(cell(1,3,P)) &
    true(cell(2,2,P)) &
    true(cell(3,1,P))

line(P) :- row(M,P)
line(P) :- column(N,P)
line(P) :- diagonal(P)

open :- true(cell(M,N,b))

index(1)      filler(x)
index(2)      filler(o)
index(3)      filler(b)
```

Using this conceptualization of states, we can define the game of Tic Tac Toe with a small set of logical sentences, as shown here. The game has thousands of states, yet it can be described with just one page of rules. A similarly parsimony is possible for other games. For example, Chess has more than 10^30 states and yet it can be described in about four pages of rules of the sort shown here. ... Let's look at each of these groups of rules in more detail.

Roles

```
role(white)
role(black)
```

We first identify the two roles in the game, viz. white and black.

Propositions

```
base(cell(X,Y,W)) :-
  index(X) &
  index(Y) &
  filler(W)

base(control(W)) :-
  role(W)

index(1)
index(2)
index(3)

filler(x)
filler(o)
filler(b)
```

Next, we define the propositions. Since there are only 29 propositions, we could do this by writing 29 ground atoms. However, we can do this more economically by writing two rules, as shown here, together with six ground atoms. An expression of the form cell(X,Y,W) is a proposition if X is an index and Y is an index and W is a role. An index is either 1, 2, or 3. A filler is either an x, an o, or a b (for blank).

Actions

```
input(W,mark(X,Y)) :-
  role(W) &
  index(X) &
  index(Y)

input(W,noop) :-
  role(W)
```

We can do the same with actions. An expression mark(X,Y) is an action for W if W is a role, if X is an index, and if Y is an index. noop is also a possible action for either player.

Initial State

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

Next, we characterize the initial state by writing all relevant propositions that are true in the initial state. In this case, all cells are blank; and the x player has control.

Legality

```
legal(W,mark(X,Y)) :-
    true(cell(X,Y,b)) &
    true(control(W))

legal(white,noop) :-
    true(control(black))

legal(black,noop) :-
    true(control(white))
```

Next, we define legality. A player may mark a cell if that cell is blank and it has control. Otherwise, the only legal action is noop.

Physics

```
next(cell(M,N,x)) :-
    does(white,mark(M,N))

next(cell(M,N,o)) :-
    does(black,mark(M,N))

next(cell(M,N,b)) :-
    does(W,mark(J,K)) &
    true(cell(M,N,b)) & M!=J

next(cell(M,N,b)) :-
    does(W,mark(J,K)) &
    true(cell(M,N,b)) & N!=K

next(cell(M,N,Z)) :-
    does(W,mark(M,N)) &
    true(cell(M,N,Z)) & Z!=b

next(control(white)) :-
    true(control(black))

next(control(black)) :-
    true(control(white))
```

Next, we look at the update rules for the game, in other words, its “physics”. A cell is marked with an x or an o if the appropriate player marks that cell. If a cell is blank and is not marked on that step, then it remains blank. If a cell contains a mark, it retains that mark on the subsequent state. Finally, control alternates on each play.

Supporting Concepts

```
row(M,W) :-
  true(cell(M,1,W)) &
  true(cell(M,2,W)) &
  true(cell(M,3,W))

column(N,W) :-
  true(cell(1,N,W)) &
  true(cell(2,N,W)) &
  true(cell(3,N,W))

line(W) :- row(M,W)
line(W) :- column(N,W)
line(W) :- diagonal(W)

open :- true(cell(M,N,b))
```

```
diagonal(W) :-
  true(cell(1,1,W)) &
  true(cell(2,2,W)) &
  true(cell(3,3,W))

diagonal(W) :-
  true(cell(1,3,W)) &
  true(cell(2,2,W)) &
  true(cell(3,1,W))
```

Before we get to rewards and termination, here are some supporting concepts. A row of marks mean thats there three marks all with the same first coordinate. The column and diagonal relations are defined analogously. A line is a row of marks of the same type or a column or a diagonal. Finally, a game is open provided that there is some cell containing a blank.

Goals and Termination

```
goal(white,100) :- line(x)
goal(white,50) :- ~line(x) & ~line(o)
goal(white,0) :- line(o)

goal(black,100) :- line(o)
goal(black,50) :- ~line(x) & ~line(o)
goal(black,0) :- line(x)


terminal :- line(W)
terminal :- ~open
```

Goals. The white player gets 100 points if there is a line of x marks and no line of o marks. If there are no lines of either sort, white gets 50 points. If there is a line of o marks and no line of x marks, then white gets 0 points. The rewards for black are analogous.

And, finally, termination. A game terminates whenever either player has a line of marks of the appropriate type or if the board is not open, i.e. there are no cells containing blanks.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Game Simulation Example

Roles

```
role(white)
role(black)
```


```
role(white)
role(black)
```


As an exercise in logic programming and GDL, let's look at the outputs of the ruleset defined in the preceding segment at various points during an instance of the game. To start, we can use the ruleset to compute the roles of the game. This is simple in the case of Tic-Tac-Toe, as they are contained explicitly in the ruleset.

Base Propositions

```
base(cell(X,Y,W)) :-
  index(X) &
  index(Y) &
  filler(W)

base(control(W)) :-
  role(W)
```

```
index(1)
index(2)
index(3)

filler(x)
filler(o)
filler(b)
```

```
base(cell(1,1,x))
base(cell(1,2,x))
base(cell(1,3,x))
base(cell(2,1,x))
base(cell(2,2,x))
base(cell(2,3,x))
base(cell(3,1,x))
base(cell(3,2,x))
base(cell(3,3,x))
```

```
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))

base(control(white))
base(control(black))
```

```
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
base(cell(1,1,b))
```

Similarly, we can compute the possible propositions. Remember that this gives a list of all such propositions; only a subset will be true in any particular state.

Inputs

```
input(W,mark(X,Y)) :-
  index(X) &
  index(Y) &
  role(W)

input(W,noop) :-
  role(W)
```

```
index(1)
index(2)
index(3)
```

```
input(white,mark(1,1))
input(white,mark(1,2))
input(white,mark(1,3))

input(black,mark(1,1))
input(black,mark(1,2))
input(black,mark(1,3))
```

```
input(white,mark(2,1))
input(white,mark(2,2))
input(white,mark(2,3))

input(black,mark(2,1))
input(black,mark(2,2))
input(black,mark(2,3))
```

```
input(white,mark(3,1))
input(white,mark(3,2))
input(white,mark(3,3))

input(black,mark(3,1))
input(black,mark(3,2))
input(black,mark(3,3))
```

```
input(white,noop)
input(black,noop)
```

We can also compute the relevant actions of the game for each player. The extension of the input relation in this case consists of the twenty sentences shown here.

Initial State

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


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


The first step in playing or simulating a game is to compute the initial state. We can do this by computing the init relation. As with roles, this is easy in this case, since the initial conditions are explicitly listed in the program.

Initial State

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

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



```
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))
```

Once we have these conditions, we can turn them into a state description for the first step by asserting that each initial condition is true.

Termination

```
terminal :- line(W)
terminal :- ~open
```

```
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))
```

Taking this input data and the logic program, we can check whether the state is terminal. In this case, it is not.

Goals

```
goal(x,100) :- line(x) & ~line(o)
goal(x,50) :- ~line(x) & ~line(o)
goal(x,0) :- ~line(x) & line(o)
goal(o,100) :- ~line(x) & line(o)
goal(o,50) :- ~line(x) & ~line(o)
goal(o,0) :- line(x) & ~line(o)
```

```
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))
```

```
goal(white,50)
goal(black,50)
```

We can also compute the goal values of the state. Since the state is non-terminal, there is not much point in doing that; but the description does give us the values shown here.

Legal Actions

```
legal(P,mark(X,Y)) :-
  true(cell(X,Y,b)) &
  true(control(P))

legal(x,noop) :-
  true(control(black))

legal(o,noop) :-
  true(control(white))
```

```
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))
```

```
legal(white,mark(1,1))
legal(white,mark(1,2))
legal(white,mark(1,3))
legal(white,mark(2,1))
legal(white,mark(2,2))
legal(white,mark(2,3))
legal(white,mark(3,1))
legal(white,mark(3,2))
legal(white,mark(3,3))
legal(black,noop)
```

More interestingly, using this state description and the logic program, we can compute legal actions in this state. The white player has nine possible actions (all marking actions), and the black player has just one (noop).

Actions

```
does(white,mark(1,1))
does(black,noop)
```

Let's suppose that the white player chooses the first legal action and the black player chooses its sole legal action noop. This gives us the dataset for does shown here.

Next State

```
next(cell(M,N,x)) :-
    does(white,mark(M,N))

next(cell(M,N,0)) :-
    does(black,mark(M,N))

next(cell(M,N,b)) :-
    does(P,mark(J,K)) &
    true(cell(M,N,b)) &
    distinct(M,J)

next(cell(M,N,b)) :-
    does(P,mark(J,K)) &
    true(cell(M,N,b)) &
    distinct(N,K)
```

```
next(cell(M,N,Z)) :-
    does(P,mark(M,N)) &
    true(cell(M,N,Z)) &
    distinct(Z,b)

next(control(white)) :-
    true(control(black))

next(control(black)) :-
    true(control(white))
```

```
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))
```

```
does(white,mark(1,1))
does(black,noop)
```

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

Now, combining this dataset with the state description above and the logic program, we can compute what must be true in the next state. For example, using the first update rule and the first does fact, we can conclude the first fact about the next state, viz. that there will be an x in cell 1,1.

Next State

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



```
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(o))
```

To produce a description for the resulting state, we substitute true for next in each of these sentences and repeat the process. This continues until we encounter a state that is terminal, at which point we can compute the goals of the players in a similar manner.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Miscellaneous

Completeness

Of necessity, game descriptions are logically incomplete in that they do not uniquely specify the moves of the players.

Every game description contains complete definitions for initial state, legality, termination, goalhood, and update in terms of the state and the does relation.

The upshot is that in every state every player can determine legality, termination, goalhood and, given a joint move, can update the state.

Of necessity, game descriptions are logically incomplete in that they do not uniquely specify the moves of the players. However, every game description does contain complete definitions for initial state, legality, termination, goalhood, and update. The upshot is that in every state every player can determine legality, termination, goalhood and, given a joint move, can update the state.

Termination

A game description in GDL *terminates* if and only if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps.

A game terminates if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps. In General Game Playing, we currently require that all games terminate in this way.

Playability

A game is playable if and only if every player has at least one legal move in every non-terminal state.

Note that in chess, if a player cannot move, it is a stalemate. Fortunately, this is a terminal state.

In GGP, we guarantee that every game is playable.

A game description in GDL is playable if and only if every role has at least one legal move in every non-terminal state reachable from the initial state. Note that in chess, if a player cannot move, it is a stalemate. Fortunately, this is a terminal state. In GGP, we guarantee that every game is playable.

Winnability

A game is strongly winnable if and only if, for some player, there is a sequence of individual moves of that player that leads to a terminating goal state for that player.

A game is weakly winnable if and only if, for every player, there is a sequence of joint moves of the players that leads to a terminating goal state for that player.

In GGP, every game is weakly winnable, and all single player games are strongly winnable.

A game is strongly winnable if and only if, for some player, there is a sequence of individual moves of that player that leads to a terminating goal state for that player. A game is weakly winnable if and only if, for every player, there is a sequence of joint moves of the players that leads to a terminating goal state for that player. In GGP, every game is at least weakly winnable, and all single player games are strongly winnable.

Obfuscation

What we see:

```
next(cell(M,N,x)) :-
    does(white,mark(M,N)) &
    true(cell(M,N,b))
```

What the player sees:

```
next(welcoul(M,N,himenoing)) :-
    does(himenoing,dukepse(M,N)) &
    true(welcoul(M,N,lorenchise))
```

Obfuscation next. In order to prevent programmers from building in specialized capabilities for specific words in game descriptions, it is common for game managers to obfuscate descriptions. All words are consistently replaced by nonsense words, as in the example shown here. The only exceptions are variables and a selection of constants common to all games, such as next, does, true, and so forth.

Prefix GDL

Syntax is prefix version of standard syntax.
Operators are renamed.
Case-independent. Variables are prefixed with `?`.

```
r(X,Y) :- p(X,Y) & ~q(Y)
```

```
(<= (r ?x ?y) (and (p ?x ?y) (not (q ?y))))
```

or

```
(<= (r ?x ?y) (p ?x ?y) (not (q ?y)))
```

Semantics is the same.

Finally, there is an issue of syntax. The version of GDL presented here uses traditional infix syntax. However, this is not the only version of the language. There is also a version that uses prefix syntax. Although some general game playing environments support Infix GDL, it is not universal. On the other hand, all current systems support Prefix GDL. Fortunately, there is a direct relationship between the two syntaxes, and it is easy to convert between them. There is just one issue to worry about.

Case Independent

Prefix GDL is case independent.

```
(<= (cell ?x ?y) (rowIndex ?x) (colIndex ?y))
(<= (cell ?x ?y) (rowindex ?x) (colindex ?y))
(<= (CELL ?x ?y) (ROWINDEX ?x) (COLINDEX ?y))
```

Mapping to/from KIF might lose case information.

```
   cell(X,Y) :- rowIndex(X) & colIndex(Y)
(<= (cell ?x ?y) (rowIndex ?x) (colIndex ?y))
   cell(X,Y) :- rowindex(X) & colindex(Y)
```

Good practice to use just one case in naming constants.
Common practice to use lower case.

The issue is the spelling of constants and variables. Prefix GDL is case-independent, so we cannot use capital letters to distinguish the two. Constants are spelled the same in both versions; but, in prefix GDL, we distinguish variables by beginning with the character '?'. Thus, the constant a is the same in both languages while the variable X in Infix GDL is spelled ?x or ?X in Prefix GDL. Unfortunately, mapping between the two can be tricky since a case-independent system might discard case information. Hence, it is good practice to use just one case in naming constants; and it is common practice to use lower case.

White Space

These are the same:

```
(cell a ?y)
( cell a ?y)
(cell a ?y )
(cell     a     ?y)
```

These are not the same:

```
(cell a ?y)
(ce ll a ?y)
(cell a ? y)
```

Finally, just to be clear on this, in Prefix GDL white space (spaces, tabs, carriage returns, line feeds, and so forth) can appear anywhere other than in the middle of constants, variables, and operator names. Thus, there can be multiple spaces between the components of an expression; there can be spaces after the open parenthesis of an expression and before the operator or relation constant or function constant; and there can be spaces after the last component of an expression and the closing parenthesis.


GENERAL
GAME
PLAYING