CHAPTER 2

Game Description

2.1 Introduction

The most significant 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.

In General Game Playing, games are defined in a formal language known as GDL. GDL is a logic programming language. (See the Appendix.) It is similar to other logic programming languages, such as Datalog and Prolog, except that (1) its semantics is purely declarative, (2) it has restrictions that assure that all questions of logical entailment for any description in the language are decidable, and (3) it includes some reserved words that specialize it for the description of games.

This chapter is an introduction to GDL and the issues that arise in using it to describe games. We start with an introduction to the game model underlying GDL; we then define the language; we look at a sample game description; and we look at the use of this description in simulating a match of the game. We then talk about additional features of games that ensure that they are interesting. Finally, we summarize the prefix syntax for GDL used in most GGP competitions.

2.2 Game Model

The GDL model of games starts with entities and relations. *Entities* represent objects presumed or hypothesized to exist in the game. *Relations* represent properties of those objects or relationships among them.

In our examples here, we refer to entities and relations using strings of letters, digits, and a few non-alphanumeric characters (e.g. "_"). For reasons described below, we prohibit strings beginning with upper case letters; all other combinations are acceptable. Examples include `x`, `o`, `123`, and `white_king`.

The set of all entities that can be used in a game is called the *domain* of the game. The set of all relations in a game is called the *signature* of the game. In GDL, domains and signatures are always finite (albeit in some cases very, very large).

The *arity* of a relation is the number of objects involved in any instance of that relation. Arity is an inherent property of a relation and never changes.

A game *schema* consists of a domain, a signature, and an assignment of arities for each of the relations in the signature.

Given a game schema, we define a *proposition* to be a structure consisting of an *n*-ary relation from the signature and *n* objects from the domain. In what follows, we write propositions using traditional mathematical notation. For example, if `r` is a binary relation and `a` and `b` are entities, then `r(a,b)` is a proposition.

The *propositional base* for a game is the set of all propositions that can be formed from the relations and the entities in the game's schema. For a schema with entities `a` and `b` and relations `p`

and `q` where `p` has arity 1 and `q` has arity 2, the propositional base is {`p(a)`, `p(b)`, `q(a,a)`, `q(a,b)`, `q(b,a)`, `q(b,b)`}.

In GDL, propositions are usually partitioned into disjoint classes, viz. base propositions and effectory propositions (more commonly called *actions*). Base propositions represent conditions that are true in the state of a game, and effectory propositions represent actions performed by game players. (Later, in order to deal with partial information, we add sensory propositions (or *percepts*) to this partition. For now, we ignore percepts.)

Before proceeding, let's look at these concepts in the context of a specific game, viz. Tic Tac Toe. As 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 relation `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 the cell in row `2` and column `3` contains an `o`. We use the unary relation `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 0-ary relation `noop` refers to the act of doing nothing.

A *state* of a game is an arbitrary subset of the game's base 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 below on the left with the set of propositions shown on the right.

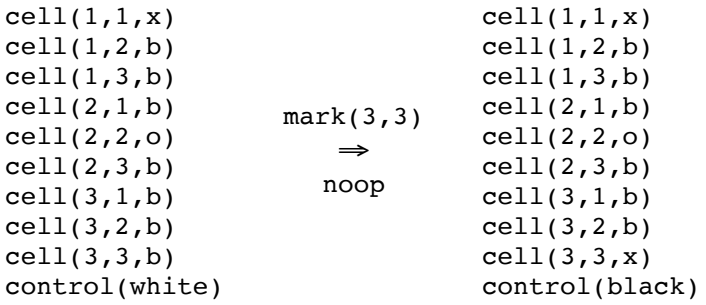
```
cell(1,1,x)
cell(1,2,b)
cell(1,3,b)
cell(2,1,b)
cell(2,2,o)
cell(2,3,b)
cell(3,1,b)
cell(3,2,b)
cell(3,3,b)
control(white)
```

On each time step, each role in a game has one or more legal actions it can perform. The actions on the left below are the legal moves for `white` in the state shown above. In this state, `black` has only one legal action, viz. `noop`, as shown on the right.

```
mark(1,2)    noop
mark(1,3)
mark(2,1)
mark(2,3)
mark(3,1)
mark(3,2)
mark(3,3)
```

A *move* in a game corresponds to a list of actions, one for each role. As a move is performed, some base propositions become true and others become false, leading to a new set of true propositions and, consequently, a new state and possibly a new set of legal actions.

For every state and every move in a game, there is a unique next state For example, starting in the state shown below on the left, if the players perform the actions shown on the arrow, the game will move to the state shown on the right.



Every game is assumed to have a unique initial state and one or more terminal states. Every state is also assumed to have a value for each player - the number of points the player gets if the game terminates in that state. For example, the state on the right above is worth 100 points to `white` and 0 points to `black`.

A game starts in the initial state. The players select and execute legal actions in that state. The game then moves on to the next state (based on the players' actions). This process repeats until the game enters a terminal state, at which point the game stops and the players are awarded the number of points associated with the terminal state.

2.3 Game Description Language

In GDL, we fix the meanings of some words in the language for all games (the *game-independent vocabulary*) while at the same time allowing game authors to use their own words for individual games (the *game-specific vocabulary*).

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. GDL has no game-independent function constants. However, there are ten game-independent relation constants, viz. the ones shown below.

`role(`*a*`)` means that *a* is a role in the game.
`base(`*p*`)` means that *p* is a base proposition in the game.
`input(`*r*`,`*a*`)` means that *a* is a feasible action for role *r*.
`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 role *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.

A GDL description is an open logic program with the following input and output relations. (1) A GDL game description must give complete definitions for `role`, `base`, `input`, `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.

We can describe these concepts abstractly. However, experience has shown that most people learn their meaning more easily through examples. In the next section, we look at a definition of one particular game, viz. Tic Tac Toe.

2.4 Game Description Example

We begin with an enumeration of roles. In this case, there are just two roles, here called `x` and `o`

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

We can characterize the propositions of the game as shown below.

```
base(cell(M,N,x)) :- index(M) & index(N)
base(cell(M,N,o)) :- index(M) & index(N)
base(cell(M,N,b)) :- index(M) & index(N)

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

We can characterize the feasible actions for each role in similar fashion.

```
input(R,mark(M,N)) :- role(R) & index(M) & index(N)
input(R, noop) :- role(R)

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

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.

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

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

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

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

next(cell(M,N,W)) :-
  true(cell(M,N,W)) &
  distinct(W,b)

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

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

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

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

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. The line relation is defined below.

```
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)
```

Supporting concepts. A line is a row of marks of the same type or a column or a diagonal. A row of marks mean thats there three marks all with the same first coordinate. The column and diagonal relations are defined analogously.

```
line(Z) :- row(M,Z)
line(Z) :- column(M,Z)
line(Z) :- diagonal(Z)

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

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

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

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

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.

```
terminal :- line(x)
terminal :- line(o)
terminal :- ~open

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

2.5 Game Simulation Example

As an exercise in logic programming and GDL, let's look at the outputs of the ruleset defined in the preceding section 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.

```
role(x)
role(o)
```

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.

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

We can also compute the relevant actions of the game. The extension of the `input` relation in this case consists of the twenty sentences shown below.

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

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

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.

```
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))
```

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.

```
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.

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

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

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

```
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)
```

Let's suppose that the `x` player chooses the first legal action and the `o` player chooses its sole legal action. This gives us the following dataset for `does`.

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

Now, combing this dataset with the state description above and the logic program, we can compute what must be true in the 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(black))
```

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.

2.6 Game Requirements

The definitions in Section 2.2 constrain GDL to game descriptions from which it is possible to compute the legal actions of all players for each state and from which it is possible to compute the next state for each state from the actions of all players. However, there are additional constraints that limit the scope of GDL to avoid problematic games.

Termination. A game description in GDL *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.

Playability. 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.

Winnability. A game description in GDL is *strongly winnable* if and only if, for some role, there is a sequence of individual individual actions of that role that leads to a terminal state of the game where that role's goal value is maximal no matter what the other players do. A game description in GDL is *weakly winnable* if and only if, for every role, there is a sequence of *joint* actions of all roles that leads to a terminal state where that role's goal value is maximal.

Well-formedness. A game description in GDL is *well-formed* if it terminates and is both playable and weakly winnable.

In general game playing, all well-formed single player games should be strongly winnable. Clearly, it is possible to generate game descriptions in GDL which are not well-formed. Checking game descriptions to see if they are well-formed can certainly be done in general by using brute-force methods (exploring the entire game tree); and, for some games, faster algorithms may exist. Game descriptions used in GGP competitions are always well-formed. However, in this book, we occasionally look at games that are not well-formed for reasons of simplicity or pedagogy.

2.7 Prefix GDL

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 are just a few issues to worry about.

The first 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.

The second issue in mapping between the formats is syntax of expressions. In Prefix GDL, all expressions are lists of components separated by spaces and enclosed in parentheses. Also, logical operators are spelled out. The following tables illustrates the mapping.

Infix GDL	Prefix GDL
`p(a,Y)`	`(p a ?y)`
`~p(a,Y)`	`(not (p a ?y))`
`p(a,Y) & p(Y,c)`	`(and (p a ?y) (p ?y c))`
`q(Y) :- p(a,Y) & p(Y,c)`	`(<= (q ?y) (and (p a ?y) (p ?y c)))`
`q(Y) :- p(a,Y) & p(Y,c)`	`(<= (q ?y) (p a ?y) (p ?y c))`

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.

Exercises

Exercise 2.1: Consider the game description shown below.

```
role(white)          next(p) :- does(white,a) & ~true(p)
role(black)          next(p) :- ~does(white,a) & true(p)
                     next(q) :- does(white,b) & true(p)
base(p)              next(q) :- does(white,c) & true(r)
base(q)              next(q) :- ~does(white,b) & ~does(white,c) & true(q)
base(r)              next(r) :- does(white,c) & true(q)
base(s)              next(r) :- ~does(white,c) & true(r)

action(a)            goal(white,100) :- terminal
action(b)            goal(white,0) :- ~terminal
action(c)            goal(black,100) :- terminal
action(d)            goal(black,0) :- ~terminal

init(s)              terminal :- true(p) & true(q) & true(r)

legal(white,a)
legal(white,b)
legal(white,c)
legal(black,d)
```

(*a*) How many roles are there?
(*b*) How many propositions are there?
(*c*) How many feasible actions are there?
(*d*) How many actions are legal for white in the initial state?
(*e*) How many propositions are true in the initial state?
(*f*) How many are true in the state that results from white performing action a and black performing action d in the initial state?
(*g*) What is the minimum number of steps this game can take to terminate?

Exercise 2.2: More questions about the game in Exercise 2.1.

(*a*) Does the game always terminate?
(*b*) Is the game playable?
(*c*) Is the game strongly winnable for white?
(*d*) Is the game weakly winnable for white?
(*e*) Is the game strongly winnable for black?
(*f*) Is the game weakly winnable for black?

Exercise 2.3: For each of the following pairs of expressions, say whether the expression on the second line is a faithful translation of the expression on the first line into Prefix GDL.

(*a*)
```
r(a,b) :- p(a) & q(b)
(<= (r a b) (and (p a) (q b)))
```

(*b*)
```
r(a,b) :- p(a) & q(b)
(<= (r a b) (p a) (q b))
```

(*c*)
```
r(x,y) :- p(x) & q(y)
(<= (r ?x ?y) (p ?x) (q ?y))
```

(*d*)
```
r(X,Y) :- p(X) & q(Y)
(<= (r ?x ?y) (p ?x) (q ?y))
```