# Classification and Solutions of General Combinatorial Games

ERIK EDIN


KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2007

# Classification and Solutions of General Combinatorial Games

E R I K  E D I N

Master's Thesis in Computer Science (20 credits)
at the School of Engineering Physics
Royal Institute of Technology year 2007
Supervisor at CSC was Fredrik Niemelä
Examiner was Stefan Arnborg

TRITA-CSC-E 2007:048
ISRN-KTH/CSC/E--07/048--SE
ISSN-1653-5715

Royal Institute of Technology
*School of Computer Science and Communication*

**KTH** CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se

**Abstract**

This master's project is an attempt to enable automatic classification and solving of combinatorial games, in the context of general game playing. The Game Description Language is used as the framework for general game playing. The classification of the combinatorial games is focused on the basic properties normal or misère form, all small games, partial or impartial games, and number of players. The methods used in making these classifications are template matching and logical deduction. This master's thesis focuses on solving only subtraction games, since they are easily analysed, while remaining interesting. Presented here is a solution, that can analyse the most common constructs, to solving a slightly generalised form of subtraction games. The classification and solving is tested on a suite of games constructed for this master's project.

# Klassificering och lösning av generella kombinatoriska spel

**Sammanfattning**

Detta examensarbete är ett försök att möjliggöra automatisk klassificering och lösning av generella kombinatoriska spel. Som verktyg för att beskriva kombinatoriska spel används Game Description Language utvecklat på Stanford. Klassificeringen av kombinatoriska spel är fokuserad på de grundläggande egenskaperna normal- eller misèrespel, "all small"-spel, partiska och opartiska spel, samt antal spelare. Metoderna som används är igenkänning från mall och logisk härledning. Detta examensarbete fokuserar på lösning av subtraktionsspel, eftersom de är lättanalyserade, men fortfarande intressanta. Här presenteras en lösning, som kan analysera de vanligaste konstruktionerna, av en något generaliserad form av subtraktionsspel. Klassificering och lösningen testades på ett antal spel, som konstruerades för detta arbete.

# Contents

# List of Figures

# 1 Introduction

This master's project can be seen as being divided into two distinct parts. The first part is the classification of general combinatorial games. In this part I attempt to enable classification of general combinatorial games, in order to facilitate selection of an appropriate strategy for a game. This is described in section 4.

The second part pertains to automatic solving of general combinatorial games. This project focuses on solving subtraction games. This is described in section 5. Subtraction games were selected since they are both an interesting and a relatively important class of games.

## 1.1 Why combinatorial games?

When using the Game Description Language for general game playing, combinatorial games are a very appropiate class of games to analyse. The reason for this is that the limitations of the Game Description Language and the characteristics of combinatorial games coincide on several points. The Game Description Language can only describe games in which both players maintain perfect information, one of the basic requirements of combinatorial games. The absence of chance in GDL also coincides with the absence of chance in combinatorial games.

Two points in which the Game Description Language and combinatorial games do not agree are the number of players and turn taking. Combinatorial games carry strict conditions as they only allow two-player games with alternating turn taking. For a language designed to describe general games, these are naturally not acceptable restrictions. However, the turn-based design of GDL makes it easy to simulate alternating turn taking. Thus, it is a simple exercise to design relations to form the general nature of GDL to the more strict nature of combinatorial games.

## Acknowledgements

I would like to thank my supervisor Fredrik Niemelä for a fun and rewarding master's project. I would furthermore like to thank Alexander Ahl, my fellow student, with whom I've spent many fun hours in the office.

# 2 Background

The following is a presentation of the fields of general game playing, and the framework for general game playing created by the Stanford logic group.

## 2.1 General game playing

General game playing is a field of research in which the aim is to enable computers to play games that they have not seen before. This in contrast with current game playing machines that are created to play one game only. The idea is that the computer player is given a formal description of the game it is to play, and proceeds to analyse the game to select an appropriate strategy.

In the early days of research on artificial intelligence it was widely believed that a good measure of intelligence was the ability to successfully play complex games such as chess. This idea was rooted in the fact that such games have long been considered to require great intelligence to master. As such, much research was focused on making game engines capable of competing against master players of each game. To some extent, this research has also been successful. As an example, in 1995 G. Tesauro [6] published an article in which he described his backgammon playing software TD-Gammon. Given limited knowledge of backgammon, he showed that a computer could successfully learn to master the game. His software TD-Gammon can rival the best players in the world. Since the software was only given very limited knowledge on backgammon before its unsupervised training, this can be seen as evidence that it learned well, and hence displayed some form of artificial intelligence. This specific program is naturally only able to play backgammon and cannot extend or teach itself to do anything else, but it demonstrated the viability of the methods it uses. Another example of a game playing machine is Deep Blue, the computer and chess engine that in May 1997 defeated the reigning world champion of chess, Gary Kasparov, in a game of chess. However, the idea that it would require true artificial intelligence to play chess effectively against a master player in the flesh was put to rest when Deep Blue defeated Kasparov. The reason for this is that Deep Blue possesses no ”real” intelligence. Deep Blue plays chess well only because very capable computer scientists and master players have hard coded knowledge of chess into a very powerful computer. It is not capable of learning from its mistakes or adapting its game play during a game.

It is from these experiences with game playing computers that the idea of general game playing began to find a foothold. One hopes that when a computer player must deal with a variety of previously unseen games, it must be able to generalize - a form of intelligence. Barney Pell is credited with introducing the idea of general game playing in his article [5]. In this article, he uses the term ”metagame” to describe what later became known

as general game playing.

## 2.2 Game Description Language

The Game Description Langugage (GDL) [4] is a language created in the Stanford Logic group aimed at describing any finite, perfect information, deterministic game, using first predicate logic. In this context, perfect information means that each player knows the full state of the game at any given time. However, the Game Description Language allows the use of simultaneous moves, and as such, a player will not always know the effect a move he makes will have on the state of the game. From this point of view, these games may not always be perfect information games. Since we will only concern ourselves with combinatorial games, where no simultaneous moves are made, this will, however, not be an issue.

The Game Description Language is a variant of the Datalog language [4], with its syntax in the form of the Knowledge Interchange Format (KIF) language. The design of GDL suggests that the creators strived for a minimal language, in order to be able to mathematically analyse its capabilities when describing games.

The specification for the Game Description Language also describes the state vector model that keeps all information on the game.

### 2.2.1 Language basics

All expressions in GDL are prefix expressions delimited by parentheses, in the form of `(`*expression-name arg1 arg2 ... argn*`)`, where *arg*i is a GDL expression, or a constant. A constant may be any alphanumerical string. The specification does not however specify an exact range of characters that may be used for constant, so different implementations may allow different forms of constants.

Variables in GDL are written as `?foo`, and may be used instead of any constant, except as the name of a relation. Every variable must be bound by other expressions, and may not be unbound and assumed to range over the values that it can take.

An expression in GDL is a statement about something that is true. Expressions often relate a name with its argument, like `(succ a b)`. Writing `(succ ?x ?y)` would then evaluate to true, if and only if, `?x` is `a` and `?b` is `y`.

The Game Description Language also uses implications where one or several conditions imply an expression. This is written as,

```
(<= (implied-relation arg1 arg2 ... argn)
    (condition-1 arg11 arg12 ... arg1k)
    (condition-2 arg21 arg22 ... arg2l)
    .
    .
    .
    (condition-m argm1 argm2 ... argmp))
```

The list of conditions is a conjunction. If one wishes to use a disjunction as a condition, one may use the `or` relation.

### 2.2.2 Language constructs

The Game Description Language has a number of built-in constructs. The `role` relation specifies the names of the players. It is used once for each player, as

```
(role left)
(role right)
```

The `init` relation specifies the initial state of the game. It is often used to initialise boards and specify which players begins. Initialising a board can look like

```
(init (cell 1 1 o))
(init (cell 1 2 x))
(init (cell 2 1 o))
   .
   .
   .
(init (cell 5 5 b))
```

In this format, the cell at coordinates (1,1) is initialised with a piece named by `o`, the cell at coordinates (1,2) is initialised with a piece named by `x`, and so on.

The `terminal` relation specifies when the game ends. As an example, this code specifies that the game ends when each heap has reached zero.

```
(<= terminal
    (true (heap1 0))
    (true (heap2 0))
    (true (heap3 0)))
```

The `legal` relation is used to specify which moves are permitted during a turn of the game. As the previous relations this relation depends on a number of conditions. The example shows a legal subtraction move.

```
(<= (legal left (move ?h ?x))
    (succ ?x ?y)
    (true (heap ?h ?y)))
```

The `does` relation is used to check what move a player has made. It is used in the form `(does` *player move*`)` where *move* is a GDL expression of a move made legal by a `legal` relation, and `player` is a valid role.

In order for the game to progress, GDL has a way to specify the state in the next turn. This is done by the `next` relation. It is used in an implication, and causes its argument relation to be true in the next turn. The syntax is as follows.

```
(<= (next (heap 1 ?x))
    (true (succ ?x ?y))
    (does left (move ?y)))
```

This `next` implication states that in the next turn, if player left makes a move `(move ?y)` for any value `?y`, and the expression `(succ ?x ?y)` is true, then set `(heap 1 ?x)` to true, for the value of the variable `?x`.

The `goal` relation is used to assign points to players. The syntax is

```
(<= (goal left 100)
    (terminal-conditions)
    (condition-1 arg1))
```

### 2.2.3 Game Manager

The Game Manager is the entity that keeps the state vector model describing the game state, and asks each player for a move. In a server/client perspective, the Game Manager is the server while the computer players are the clients. It is the responsibility of the Game Manager to keep track of the players, and to provide them with the game description.

In each turn of the game, the Game Manager asks each player in the game for a valid move. There is no mechanism available for only asking a specific player for a move. Each player must have a valid move in every turn of the game. In section 2.2.4 we discuss briefly how to achieve the effect of a turn taking game.

The state vector model is a list of true statements about the game. In each turn of the game, when all players have made their move, the next state vector model is calculated, using the logic expressions in the game description. This is called the transition function. The state vector model is the entire model of the game.

The transition function must be completely defined. That is, states are not implicitly kept in the state vector model if nothing affects them. As an example, imagine a chess board. Say that white moves a pawn from b1 to b2, thus not affecting any other piece. It is not simply sufficient to say that

the position b1 is now blank, and b2 contains a pawn. One must also specify that each position on the board which is not affected now contains the same piece as it did on the last turn. This is the Frame Problem, a well known problem in AI research.

### 2.2.4 Language design

The Game Description Language can describe a large variety of games because it does not make many assumptions on the type of gameplay. It supports any number of players, and is not limited to alternating turn-taking games. The design of the state vector model kept by the Game Manager does, however, operate in turns, by design. In a turn, each player must make a move. In a game where players make move alternately such as in chess this is modeled by providing a move without any effect, on the other player.

# 3 Combinatorial games

Combinatorial games are characterised by being finite, deterministic two-player games. As opposed to classical game theory, in which games are often games of chance and hardly ever games in the recreational sense, combinatorial game theory describes games which are in fact recreational, but analysed from the perspective of mathematics. The deterministic nature of the games, which means there is no chance involved in the game, permit us to try to find not only optimal strategies, but perfect strategies. The distinction is that optimal strategies, in the sense of classical game theory, let us play the game as well as the rules let us, but may not always lead to the desired outcome, since there is chance involved. Perfect strategies, on the other hand, in the sense of combinatorial game theory, always lead to the desired outcome. Please note, however, that the desired outcome may be 'losing', if there is no possible way of winning.

The only possible outcomes for a player in a strictly combinatorial game is winning or losing. One does not usually analyse games with the possibility of a draw because it doesn't fit cleanly into the theory, but the literature does make some mention of these types of games. The theory of combinatorial games does not include the concept of score, although scores are very common in recreational (i.e. non-theoretical) combinatorial games. The theory simply states that a win is a win, regardless of how 'close' the losing player was to a win. For the purposes of this master thesis, we will only concern ourselves with games conforming to a strict definition of combinatorial games.

A games ends when one player cannot make a move. In the so called *normal* form of combinatorial games, a player wins when his opponent cannot make a move. In *misère* form, a player wins when he himself cannot make a move.

A combinatorial game can be considered solved when there is a known efficient way of deciding, for any given state of the game, whether the next player to move will win, or the previous player that made a move will win, assuming perfect play.

Throughout the literature and the theory of combinatorial games the two players of the game are called Left and Right, by convention. The somewhat unusual and unintuitive names much simplifies the notation of the number theory of games, of which an introduction may be found in [1].

## 3.1 Number theory on games

A number system has been developed by John Conway [3] around the notion of a game to allow us to do much more sophisticated analyses. This number theory allows us to add, subtract, compare, and solve games, for which we have calculated the value. For a more complete discussion of this number

system, see [1].

The value of a game tells us the number of moves to Left's advantage. This, however, should not be taken to imply that all games have an integer value. There are games with values of dyadic rationals, and certain games can be constructed to take any real value. This number system is not, however, completely analogous to our ordinary number system. Multiplication and division, in general, lack any meaningful interpretation and many numbers lack common and natural properties. Some numbers cannot be compared to other numbers, because their value is ill-defined. Other numbers can be shown to be greater than zero, but smaller than any strictly positive real number. Because of the unusual characteristics of this number system Donald Knuth suggested that these numbers be called *surreal*.

According to who will win the game, assuming perfect play, combinatorial games can be divided into four different classes.

1. The player that makes the opening move loses.
2. Left player wins.
3. Right player wins.
4. The player that makes the opening move wins.

Games of type 1 where the player that makes the opening move loses are given the value 0, for reasons that will become apparent later. Games of type 2 where the Left player always wins assuming perfect play are taken to be $> 0$ in this number system. Games of type 3 where the Right player always wins, assuming perfect play are taken to be $< 0$, symmetrically. The fourth class of games where the player that makes the opening move wins does not have any counterpart in our ordinary number system. These games are denoted by the *nimber* $*$, and have some unusual characteristics. See 3.4 for more information on nimbers.

It must be noted that when speaking of a combinatorial game, each new state of the game can be viewed as a separate game. For example, let $G$ be the game of Go, with no stones on the board. Let the first player place one stone on the board. Now, one may view this new board state as a game in itself, and call it $G_1$, for instance. $G_1$ is the same game as $G$ in the sense that they share the same set of abstract rules, but are different in the sense that they do not share the same set of possible next states. It is the latter interpretation that will be of use in the next section.

### 3.1.1 Operations on games

As games may be regarded as numbers, according to this number system, one may also perform operations on them. Two of the most important operations

are addition and subtraction. Multiplication and division of games does not, in general, have any meaningful interpretation.

Addition of games is done by placing two games beside one another. When a player is about to make a move, he then decides in which game to make the move, and then makes it. The other player may then, in the same way, choose which game to make a move in. He does not need to make his move in the same game as the previous player. The result of the addition is another game which ends when both constituent games has ended. There are other forms of addition, but they are beyond the scope of this thesis.

Negation is an important operation, used when subtracting games. The negation of a game $G$ is defined as the game $H$, for which it is true that $G + H = 0$. That is, when playing $G$ and $H$ beside one another, the player that makes the first move will lose, assuming perfect play. The negation of $G$ is written $-G$.

Subtraction is done by adding the negation of game to another game.

See [1] and [2] for further reading on operations.

## 3.2 Nim

Nim is a simple game played with heaps of beans. Each player takes turns in removing a number of beans from a single heap. A player must remove at least one bean from a heap, but may remove as many as he wants from one single heap. He may choose which heap to remove from freely. If he wants, he may remove all beans from the heap. The game ends when each heap is empty, since a player can no longer make a valid move.

Firstly, one may observe a few simple strategies. If there is only a single heap left, the next player will remove the entire heap, and consequently win. If there are only two heaps, of different sizes, the next player will remove the number of beans from the larger heap that make the two heaps the same size. The opponent will then have to remove beans from one heap, making them different sizes again. Applying this argument repeatedly will lead to a win for the player that equalizes the heaps, because the opponent will eventually have to make one heap empty.

## 3.3 Impartial games

Impartial games are games where both players have the same possible moves in a given state. Chess, for instance, is partial, because a player can only move pieces of his own color. Nim, on the other hand, is impartial, since both players have the same set of moves available.

Impartial games are especially important, because all impartial games can be regarded as, and played perfectly as, the game Nim. It does not, however, tell us how we can play the game, as if it were Nim. To do this, we must find the so called Sprague-Grundy function of the game.

The Sprague-Grundy function is a function that assigns a nimber to every position of an impartial game. Most commonly, this function is described as an explicit formula, shown correct by mathematical analysis. This is obviously an impractical way of deciding a Sprague-Grundy function in the context of general game playing, as computers do not have the required reasoning capabilities. Instead, one may use a more useful method of deciding the Sprague-Grundy function. This method utilizes the game tree, and assigns a nimber to each node of the tree according to the mex-rule. The mex-rule is the *minimally excluded number* rule. It simply states that the minimally excluded number of a finite set of integers $G$ is the smallest integer $x \geq 0$, such that $x \notin G$. Very large game trees may of course make this infeasible.

## 3.4 Nimbers

Nimbers are surreal numbers that represent the value of a Nim heap. The nimber $*n$ is the value of a single Nim heap of size $n$. The notation is consistent with the rest of the number system, as nimbers are merely several $*$ added together.

Nimbers can be added by the so called Nim Addition Rule, or more simply by nimber addition. Adding nimbers $*n_1, *n_2, ..., *n_k$ represents the value of a game of Nim with $k$ heaps of sizes $n_1, n_2, ..., n_k$. The power of Nim addition lies in the fact that one can see who will win the game directly from the Nim value. If the game has value $*0$, then it is a zero game, and the first player to make a move will lose, assuming perfect play. If the game has value $*m$ for $m \neq 0$, then the first player to make a move will win, assuming perfect play.

Adding two nimbers $*n_1$ and $*n_2$ is done by bitwise addition modulo 2. That is, let the binary representation of $n_1$ be $b_1^1 b_2^1 ... b_m^1$, and the binary representation of $n_2$ be $b_1^2 b_2^2 .. b_m^2$. Here we assume that $m$ is large enough to contain both binary representations. If $*n_3 = *n_1 + *n_2$, and the binary representation of $*n_3$ is $b_1^3 b_2^3 ... b_m^3$, then we have that $b_i^3 = b_i^1 \oplus b_i^2$, where $\oplus$ is addition modulo 2.

For further information on Nim, nimbers, and a proof of the claim that Nim addition solves Nim, see [1].

## 3.5 Subtraction games

Subtraction games are similar to Nim. There are a number of heaps and a player makes his move by removing a number of beans from a heap. However, in subtraction games a player is not allowed to remove any number of beans. There is a set of numbers specifying how many beans the player is allowed to remove from a heap. This set is called the *subtraction set*. For example. if the subtraction set is 2, 5, 6, then a player may remove 2, 5, or 6 beans

from a heap.

A subtraction game is defined by the initial sizes of the heaps, and the subtraction set. Since it is an impartial game, it can be solved by applying the Sprague-Grundy function.

## 3.6 Perfect strategies

The reason one can make the distinction of games into the four outcome classes is that combinatorial games are deterministic, perfect information, two-player games. When a game has been classified according to these classes, with a perfect strategy made explicit, the game may be considered solved. Two people aware of the perfect strategy of the game may then set up the game, decide who begins, and then proclaim one of the players the winner, without ever making a move.

# 4 Automatic classification

Classification of games based on their properties is the most basic and important method for analyzing games, and the strategies used to play them. In the context of general game playing, this is made more difficult, as we do not know in advance what game we will be playing. Hence, we need methods for automatic classification of a general game so that appropriate strategies can be selected to play the game.

This master's project is limited only to classification of combinatorial games only. As they are finite, deterministic in their nature, and often very simple, combinatorial games are very well suited for automatic analysis and subsequent playing. Games with chance, infinite boards, or incomplete information are beyond the scope of this project.

As the Game Description Language uses first order logic, it is vital to have a way of infering logical statements when analysing a game. See section 6 for information on the specific logic engine used in this master's project.

## 4.1 Categories

The categories chosen for this project represent the most basic classifications of games, as they are the most important for choosing a proper strategy.

The categories chosen are

- Two-player games, or not.
- Normal or misère form, or other.
- Partial and impartial games.
- "All small" games, or not.

A game described by the Game Description Language is here considered to be combinatorial if and only if it is a two-player game, and in normal or misère form.

Classification of games into two-player games, or more or less players, is trivial, but also very important. This will help us in identifying combinatorial games as well as providing information for the analysis of subsequent classes. Here we will not consider games with more than two players that are equivalent with two player games.

Normal and misère forms of games specify a type of condition that needs to be fulfilled in order to win the game. In its own right, this is important, but which form the game is in provides more information than that. It may be considered counterintuitive, but nevertheless true, that misère games are much harder to analyze than games in normal form. Thus, if a game is classified as being in misère form, one may simply choose to abandon any attempt to solve the game. For a more in depth discussion of this, see [2].

Classifying according to partial and impartial games is also a very broad and extremely important classification. Showing that a game is impartial is very helpful as the game then is a variant of Nim.

The category of ”all small” games is the most specific category in this project. It is somewhat related to the category of partial and impartial games, as impartial games are well known to be ”all small”. This category is important because the TERMOSTRAT strategy is used for games with hot positions, and in all small games, there are no hot positions. The TERMOSTRAT strategy is an imperfect, but in practice very good strategy. See [2] for more information.

## 4.2 Deciding who is in control

All combinatorial games are alternating turn-based games, which means that a formal description of a game must keep track of which player may move, in any given turn. The Game Description Language was designed with a broader class of games in mind such as games with simultaneous moves, which, unfortunately, means that it does not specify a built-in way of defining which player's turn it is. Since deciding whose turn it is is a very central question in the analysis of combinatorial games, we must, for every game we wish to analyse, find the method the particular game uses to decide whose turn it is. Fortunately, a majority of the games written in the Game Description Language that are currently available, use similar or exactly the same method. The most common way of specifying turn-taking is defining a relation, e.g. `(control left)`, which states that it is player `left`'s turn to move. To alternate the turn, one simply makes use of the `next` relation, specifying that it is other player's turn to move next turn. The full code for doing this then becomes,

```
(role left)
(role right)

(init (control left))

(<= (next (control right))
    (true (control left)))

(<= (next (control left))
    (true (control right)))
```

Since the name `control` of the relation is merely a common way of doing it, and not a specified standard, and since names of relations are garbled in competitions, one cannot simply search for a relation named `control`. One can however search for this particular pattern of alternating `next` relations with an accompanying relation specifying a player, and thereby correctly

identify this relation as the method of defining turns for the players for a great majority of games.

Furthermore, since all players must make a move in every turn, as specified by the Game Description Language and its use, the player whose turn it is *not* to move, must have a legal move. This is most commonly solved by defining a move, often named `noop`, which does nothing.

```
(<= (legal left noop)
    (not (true (control left))))

(<= (legal right noop)
    (not (true (control right))))
```

No `next` relations are necessary here, since this move will have no effect.

## 4.3 Normal and misère forms

When playing a game in normal form, the player last to make a move wins. More to the point, the player whose turn it is when the terminal condition is met, has lost, since he has no available moves. The problem of determining if a game is in normal or misère form is therefore reduced to the problem of determining which player's turn it is for all goal conditions.

The Game Description Language specifies that to each player the game awards between 0 and 100 points. It is natural to assume that being awarded more points represents winning and less points represent losing, although the opposite assumption is not unnatural at all. The goal conditions will look like

```
(<= (goal 100 left)
    .
    .
    (true (control right))
    .
    .)
```

or equivalently,

```
(<= (goal 100 left)
    .
    .
    (not (true (control left)))
    .
    .)
```

In this second example, the logic engine used may need to be supplied some auxiliary rules that stipulate that if it is not `left`'s turn, then it is `right`'s

turn. One may simply supply these two rules to model this behaviour. This code is not correct GDL, but is correct KIF, the language that GDL is partly based on.

```
(<=> (true (control left)) (not (true (control right))))
(<=> (true (control right)) (not (true (control left))))
```

To infer which player is in control at the end of the game, what is most simple is change the direction of the implication arrow of the `goal` relation. This is possible since the conditions of the implication are implicitly conjuctions. When all reversed implications have been stated, one simply specifies that each `goal` implication has been achieved, one by one. For each such achieved implication, one then asks whose turn it is to move.

## 4.4 All small games

A small game is a game whose value is $*$, or a multiple thereof. These games do not provide even a single moves advantage for any one player. An ”all small” game is one in which it is true that for any fixed game state, if one player has a possible move, then so does the other. They are called ”all small”, since all positions in the game have small values.

To determine if a game is ”all small”, we first note that each legal move has a number of conditions associated with it. For a player, and if a game is ”all small”, the conditions of one of its moves must imply the conditions of some move for the other player. If there is such an implied move for all players, then the game is all small. One must note, however, that a most common condition for the legality of a move is that it is the player in question, who has the turn. Since we are not interested in turn taking when analysing whether a game is all small or not, we must disregard this specific condition. In the Game Description Language, this is most easily done by redefining the `control` relation to be true for all players.

Pseudo code for determining if a game is ”all small” or not is in figure 1.

## 4.5 Partial and impartial games

All small games are relatively easy to classify, as one looks only at the conditions of moves, and determine if they are the same. Classifying games as being partial or impartial has similar objectives, but is much harder. The reason for this is that when determining whether a game is partial or impartial one must compare the effects of moves. Another problem is that this analysis must be done for all game states. For this reason, this master’s project only performs a simple and highly unreliable test for partiality. The method consists of making a lexical comparison of the moves involved, as well as the conditions for those moves. The assumption is that moves and

```
IS-ALL-SMALL-GAME(G)
 1  for all moves D_L for left
 2  do
 3      Let C denote the conditions of move D_L's legality in game G
 4      b ← false
 5      for all moves D_R for right
 6      do
 7          Let C' denote the conditions of move D_R's legality in game G
 8          if conditions C imply C'
 9            then b ← true
10      if b = false
11        then return Not all small
12  return All small
```

Figure 1: Pseudo code to determine if a game is "all small" or not.

conditions carry different names for different purposes and similar names for the same purposes.

### 4.5.1 Weak comparison

The lexical comparison needs to disregard differences in variable names in GDL expressions, as the names themselves carry no meaning. Therefore, comparing two expressions is done by a so-called *weak* comparison. This weak comparison is a lexical comparison, with the reservation that differences in variable names is ignored. Future work may choose to extend this comparison slightly by saying that a variable name may also be equal to a constant.

The motivation for using this weak comparison is one of pragmatics. It has proven to be useful in many parts of the analysis of games, and has also proven to be sufficient for the games tested in this project. However, there are no theoretical grounds for which to motivate its existence or usefulness.

### 4.5.2 Determining partiality

In figure 2 and 3 there is pseudo code for determining partiality of a game as described in the previous sections.

```
IS-IMPARTIAL(G)
 1  for all moves D_L for left
 2  do for all changes S_{D_L} that D_L makes to the state
 3     do hasEquivalentMove ← FIND-EQUIVALENT-MOVE(right, S_{D_L})
 4        if hasEquivalentMove = false
 5           then return PARTIAL
 6  return IMPARTIAL
```

Figure 2: Pseudo code to determine if a game is impartial or not.

```
FIND-EQUIVALENT-MOVE(player, S)
 1  for all moves D for player
 2  do for all changes S_D that D makes to the state
 3     do if comparesWeakly(S, S_D)
 4           then return true
 5  return false
```

Figure 3: Pseudo code to find an equivalent move available to a specified player.

# 5 Solving subtraction games

This work is done from the perspective of game theory, and thus seeks to find an exact solution to a game. Because of this, we do not use artificial intelligence techniques to find heuristics or seemingly advantageous paths in the game, but determine the structure of the game and perform game theoretic analysis on it.

Solving general combinatorial games is a very hard task to undertake, so it is necessary to restrict the types of games one attempts to solve. Subtraction games constitute a relatively simple, but yet interesting class of games.

The approach used here to solve general games can be viewed as having two distinct parts. The first part is the general game analysis in which the rules of the games are analysed to determine what type of game it is, and how the game is played. The second part is the game theoretic analysis in which one uses the knowledge gained from the first part, on how the game is played, to determine a solution to the game.

A solution is a way of deciding, for each position played, what moves are perfect. This knowledge would allow a player to play the game in a perfect way. It is not necessary, and perhaps often infeasible, to construct an explicit solution in which each possible position in the game is listed and paired with the perfect moves from that position. Instead, one may choose to represent the solution as an oracle, that takes as input a game position, and responds with one or several perfect moves. The oracle can then avoid calculating perfect moves for each possible game position, and instead do only those calculations that are necessary. For board games, in which the number of possible positions can be very large, this may be the only feasible option.

## 5.1 Subtraction games

When solving general subtraction games, this master's project uses a slightly generalised notion of subtraction games. In a combinatorial subtraction game, there exist a number of heaps, and there is a subtraction set, defining the set of legal moves, which is the same for all heaps. However, in this project, each heap may have its own subtraction set, and none of the subtraction sets need to be equal. The reason for this slight generalisation is that it can be done without making the analysis significantly more complex. This is discussed further in section 5.4.1.

## 5.2 General game analysis

The approach used here for general game solving is that of recognizing a set of games which we know how to solve. In this master's project, we only solve subtraction games. This approach is more similar to the expert system of the

chess playing Deep Blue than the learning system of TD-Gammon, which was previously highlighted as a good example of an artificial intelligence system. On the other hand, while Deep Blue can only play chess, this system can solve a set of games.

### 5.2.1 Recognizing arithmetic

Neither the concept of integers, nor the concept of arithmetic is built into the Game Description Language. Because of this fact each game designer must implement arithmetic using the facilities in GDL, when needed. Naturally, subtraction games need a way to model subtraction. The standard way of modeling arithmetic in GDL is a relation most often called `succ`. What this relation usually does is define a total ordering of constants so that they may be used for comparison. The way one defines the `succ` relations is by defining, for a constant, what constant follows in the total ordering. In the Game Description Language, this is written in the following way,

```
(succ 0 1)
(succ 1 2)
(succ 2 3)
  .
  .
  .
(succ 8 9)
(succ 9 10)
```

It is worth noting that the use of numbers in this example is for convenience only. The numbers are in fact interpreted as string constants, without any meaning. Hence, a game designer may choose to use arbitrary constants in place of numbers, such as

```
(succ a k)
(succ k c)
(succ c q)
```

The specification of the Game Description Language states that each game must be representable by finitely many relations. It specifically forbids a game designer from using unbounded recursion. Also, since constants written as numbers carry no meaning, one must specify an explicit name for each element in the ordering. Because of this, using recursion to define a total ordering of numbers is not possible. This means that a game designer must explicitly state the ordering of the elements, and also their constant names, and hence that the order must be finite. This is why the method shown above is considered to be the most natural way of defining arithmetic.

Automatic recognition of such relations is based on the fact that they define a finite total ordering (with a beginning and an end), with connected constants. The following conditions define these relations.

- It carries two arguments
- Relates at least three pairs of values
- There is exactly one constant `s` for which there is no `s'`, such that `(succ s' s)`.
- There is exactly one constant `e` for which there is no `e'`, such that `(succ e e')`.
- For any constant, call it `b`, except for `s`, there is exactly one constant `a` for which it holds that `(succ a b)`.
- For any constant, call it `b`, except for `e`, there is exactly one constant `c` for which it holds that `(succ b c)`.

The conditions that require that it relates at least three values is used to filter relations that only relate one or two values, and trivially satisfies the other conditions. The rest of the conditions state that the ordering has a beginning and an end, and that every element has a unique element in front of it and a unique element after, except start and end, respectively.

## 5.3 General subtraction game analysis

The core components in a subtraction game are the piles and the subtraction set. To continue with solving the subtraction game, one needs to determine the following things.

- The number of piles.
- The size of each pile.
- The subtraction set for each pile.

The difficulty here, as in all general game analyses, is to determine how the author of the game description chose to model the different components of the game.

### 5.3.1 Models of heaps

A heap is modeled by assigning a heap identifier with the current size of the heap. In the simplest case, this would look like `(heap 7)`. The heap identifier is the name `heap` of the relation, and 7, of course, represents the size.

When dealing with subtraction games with several heaps it may be easier to create an indexed heap relation instead. With this method, one instead gives the heap relation two arguments: the index of the heap, and its size. This is similar to the technique for creating boards. This will look like

`(heap 3 7)`, which refers to the third heap, and its size 7. Of course, there is no reason why the index must be the first argument, and the size to be the second. However, most games using boards have been written so that each board position is referenced by their coordinates first, and the piece on the board position last, like `(cell ?x ?y pawn)`. This indicates that it is natural for people to write indexed relations in this order.

When determining what heaps are present in a game, observe that the heap relations must be initialised by an `init` relation so that each heap has an initial size. Thus, to determine the heap structure, one looks at all initialised relations, and determines if they match any of the templates `(<name> <integer-size>)` or `(<name> <integer-index> <integer-size>)`. To do this, one must first have analysed the relation defining arithmetic in order to recognize integers.

In the implementation of this type of modeling of heaps, the software recognizes the heap relations by the templates described above, and stores them in a structure. This structure allows for retrieving objects representing heaps based on a relation, bearing reference to those heaps. In this way, it is simple to later assign subtraction sets on a per-heap basis.

### 5.3.2 Determining subtraction sets

The subtraction set is modeled by the legal moves in the game. There are two obvious ways of modeling a move in a subtraction game. The first one is to specify by what amount to subtract a heap. Call this a *relative* subtraction move. The second way is to specify to what value one wishes to subtract to. Call this an *absolute* subtraction move.

For each legal relation, collect all next relations relevant to this move. Determine what variable represents the current size of the heap. In an absolute subtraction move, this must be present in the conditions of the `legal` relation. Call this variable `?source`. In the `next` relation, one will find the variable representing the size of the heap after the move has been made. Call this variable `?target`. The goal is to create a graph for a `legal` relation and all its relevant `next` relations, representing all possible subtractions. A node in this graph will be a variable name, and an edge in this graph, from, say, node `?a` to `?b?`, will be created after finding a `succ` relation `(succ ?b ?a)`. Note the order of these variables.

The following piece of code is from the game 21 and serves to exemplify how a subtraction graph is built.

```
(<= (legal ?player (move ?x))
    (true (control ?player))
    (open)
    (pmove ?x))
```

```
(<= (pmove ?x)
    (true (value ?y))
    (succ ?x ?y))

(<= (pmove ?x)
    (true (value ?y))
    (succ ?x ?a)
    (succ ?a ?y))

(<= (pmove ?x)
    (true (value ?y))
    (succ ?x ?a)
    (succ ?a ?b)
    (succ ?b ?y))

(<= (next (value ?x))
    (does ?player (move ?x)))
```

For each `legal` relation, build a list of all `succ` relations that are conditions for the `legal` relation in question. In the example above, this would include all `succ` relations in the three `pmove` implications. One must take special care with the relation `(true (value ?y))`, as this represents the source value. For each occurrence of the source variable name, rewrite it to use the `?source` variable name instead. One would then expect the above example to look like

```
(<= (legal ?player (move ?x))
    (true (control ?player))
    (open)
    (pmove ?x))

(<= (pmove ?x)
    (true (value ?source))
    (succ ?x ?source))

(<= (pmove ?x)
    (true (value ?source))
    (succ ?x ?a)
    (succ ?a ?source))

(<= (pmove ?x)
    (true (value ?source))
    (succ ?x ?a)
    (succ ?a ?b)
```

```
    (succ ?b ?source))

(<= (next (value ?x))
    (does ?player (move ?x)))
```

If the code is written with *relative* subtraction moves, then the `succ` relations will be found in the `next`-relation connected with the `legal` relation in question. The following method will be exemplified by extracting the `succ` relations from a `legal` relation, but will work in the same way if extracted from a `next`-relation. The software built for this master's project attempts to interpret moves in both ways, and determines which way provides a valid subtraction move.

In order to create the list of `succ` relations one must take care with clashing variable names. The argument in each implication must be renamed to the argument in the use of this relation. In the `legal` relation

```
(<= (legal ?player (move ?x))
    (true (control ?player))
    (open)
    (pmove ?x))
```

the argument to `pmove` is `?x`. In each implication of `pmove` the argument must then be replaced. In this example, however, the argument name in the `pmove` relation is the same as all the arguments in the `pmove` implications. All variable names internal to each implication must be renamed to have unique names in the current context.

After the list has been constructed, one can build a subtraction graph from it. The graph is built by simply going through each `succ` relation, create a node for each argument, if they do not already exist in the graph, and connect and edge from the second argument to the first. In the above example, the list would look something like

```
(1)  (succ ?x ?source)
(2)  (succ ?x ?1_a)
(3)  (succ ?1_a ?source)
(4)  (succ ?x ?2_a)
(5)  (succ ?2_a ?3_b)
(6)  (succ ?3_b ?source)
```

Here, unique names for internal variables were created by prepending the variable name with a serial number and an underscore. On line 1, the `succ` relation represent the possible move of subtraction by one. That is, the target value `?x` is immediately followed by `?source` in the total ordering. One line 2 and 3, the `succ` relations represent the possible move of subtracting by two. The target value `?x` is followed by a variable `?1_a`, which in turn is followed by `?source`. Subtraction by three is similarly represented by lines 4-6.

### 5.3.3 Subtraction graph

Another way of representing the list of collected `succ` relations is by the *subtraction graph* in Figure 4.

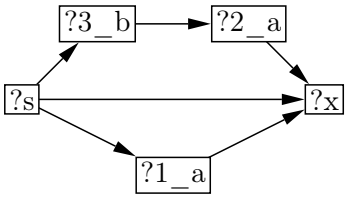


Figure 4: Subtraction graph for subtraction set $\{1, 2, 3\}$

This graph represents the relationships of all variables, as determined by the collection of `succ` relations. From a graph like this it is possible to do a search and determine all the distances of all paths from the source `?s` to the target node `?x`. These distances are then the different subtraction amounts made possible by this legal relation. In this example, it is evident that the distances are 1, 2, and 3. The subtraction graph is built as a way to succinctly represent subtraction moves. To determine what moves are encoded in this graph, one must determine all distances from the source node to the target node. To do this, keep a set of distances for each node. Do a depth first search, beginning at the source node, and for each node, add its parents' distance plus one. The target node will then have a set containing the distances of all paths from the source node. This is the subtraction set. The pseudo code in figure 5 is a simple search algorithm for doing this.

```
AllPathsDistance(G, s, t)
1  distances[s] = 0
2  stack.push(s)
3  while stack ≠ ∅
4  do m = stack.pop()
5      for all neighbours n to m
6      do for ∀d ∈ distances[m]
7          do distances[n] = distances[n] ∪ {d + 1}
8          stack.push(n)
9  return distances[t]
```

Figure 5: Pseudo code for determining all distances from a source to a target node.

```
SPRAGUE-GRUNDY(k)
1  nimseq[0] ← 0   # nimseq[0] ← 1   for misere games
2  for i ← 1 to k
3  do for  each s in S
4      do if i − s ≥ 0
5            then M ← M ∪ nimseq[i − s]
6      nimseq[i] ← mex(M)
7  return nimseq
```

Figure 6: Pseudo code for the Sprague-Grundy function of a subtraction game.

## 5.4 Game theoretic analysis

Once a game has been recognized, and all its components found and analysed, one may start the game theoretic analysis. It is the goal of this analysis to calculate a solution to the game so that one may choose advantageous moves. In combinatorial impartial games, the concept of $\mathcal{P}$-positions and $\mathcal{N}$-positions provides one basic way of choosing perfect moves. In case one needs to combine several such games, or combine several parts of games, $\mathcal{P}$-positions and $\mathcal{N}$-positions do not carry enough information, and so one needs to use the Sprague-Grundy function instead.

### 5.4.1 Subtraction game theoretic analysis

The game theoretic analysis of subtraction games is very simple. Once the general game analysis has determined the subtraction set for each pile, a simple application of the Sprague-Grundy function solves the game.

First, the analysis must be given information on the form of the game, that is, if it is in normal or misère form. This problem is in the domain of classification of combinatorial games, described in previous chapters.

When form and subtraction sets are known, the Sprague-Grundy function can be applied to each heap. The Sprague-Grundy function will assign a nimber, or Nim value, to each possible size of the heap. A player can then ask for the Nim value of a specific position in the game. A position in a subtraction game is the current sizes of all heaps. If there is more than one heap, the oracle will add the nimbers of each heap, and return the result to the player.

For a game in normal form, and a heap of initial size $k$ the application of the Sprague-Grundy function works as in figure 6. Let $nimseq[i]$ be the nim value of the heap at size $i$. The goal of the Sprague-Grundy function is to fill in this structure from $i = 0$ to $i = k$. Let $S$ be the subtraction set, and let mex(M) calculate the minimally excluded number of set M.

# 6 Implementation

In this section, the implementation of the methods of classification and solving of combinatorial is discussed.

## 6.1 Grammar of the Game Description Language

In the specification of the Game Description Language no formal grammar is provided. This means that different implementations of software using GDL may use slightly different grammars. Therefore, the grammar used for this master's project is presented as an appendix. This grammar is quite specialised and may not be suitable for general use.

## 6.2 Game suite

The game suite used to test the methods and software in this master's project needed to be created, since none of the available games written in the Game Description Language were appropriate combinatorial games. The suite consists of seven different types of games, some with several variants. The games are

- SG{1,2,3} is a subtraction game with a single pile of initial size 21, and with the subtraction set $\{1, 2, 3\}$. There are two flavors of SG{1,2,3}. One is in normal form, and the other is in misère form.
- "SG{1,2}{1,5} single" is a subtraction game with two heaps of initial sizes 7 and 11. Here, the first heap of initial size 7 has the subtraction set $\{1, 2\}$, while the second heap has the subtraction set $\{1, 5\}$. This is a game which tests the generalisation of subtraction games.
- Clobber is a popular combinatorial game. It is played on a board with black and white stones. A player makes a move by taking a stone of his own color and placing it on an adjacent (not diagonally) cell, where there is a stone of the opposite color. The stone of the opposite color is then removed from the board. This game is *partial*, but *all small*. The current implementation is in misère form.
- Kayles is played with a row of *skittles*. A player makes a move by knocking down one skittle, or two adjacent skittles. This game is impartial, and in normal form.
- Domineering is played on a board with domino pieces. One player makes his moves by placing the domino pieces horizontally along the board, always covering two adjacent positions. The other player makes his moves by placing the domino vertically along the board, also always covering two adjacent positions.

- Fox and geese is played on a 8x8 board. One player controls the fox, which starts in the upper half of the board. The other player controls four geese, whose objective it is to capture the fox. All pieces move diagonally, one square per turn, and one piece per turn. The fox is captured when the geese blocks his only possible moves. This game is partial and not all small.

- Turning turtles is a game in which there is a line of turtles, all on their feet. The objective is to turn all turtles onto their backs. The rules are that a player must flip a turtle from his feet to his back, and may optionally choose to flip any turtle, to the right of the first turtle. The player that first cannot flip a turtle onto his back, because all turtles already are on their backs, loses.

The game codes for these games are presented as appendices.

### 6.2.1 Classification of the game suite

When presented with these games, the classification software gives these results. In table 1, the games SG{1,5,7} and SG{3,5,7} are considered to be

Table 1: Classification of games in the test suite

| Game | Players | Form | Partiality | All small |
|---|---|---|---|---|
| Clobber | 2 | Misère | Impartial | All small |
| Kayles | 2 | Normal | Impartial | All small |
| Fox and geese | 2 | Unknown | Partial | Not all small |
| Turning turtles | 2 | Unknown | Impartial | All small |
| Domineering | 2 | Normal | Partial | Not all small |
| SG{1,2,3} (normal form) | 2 | Normal | Impartial | All small |
| SG{1,2,3} (misère form) | 2 | Misère | Impartial | All small |
| SG{1,2}{1,5} single | 2 | Normal | Impartial | All small |
| SG{1,2,5} (and variants) | 2 | Normal | Impartial | All small |

variants of SG{1,2,5}, since very little code differs between them. They have the exact same classifications.

### 6.2.2 Misclassifications

Clobber was here misclassified as *impartial*, but correctly classified as *all small*. As evidenced by the following code, this implementation of Clobber uses a trick, in which the name of the player becomes a marker in a board position.

```
(<= (next (cell ?fx ?fy blank))
    (does ?p (move ?fx ?fy ?tx ?ty)))
```

```
(<= (next (cell ?tx ?ty ?p))
    (does ?p (move ?fx ?fy ?tx ?ty)))
(<= (next (cell ?x ?y ?v))
    (does ?p (move ?fx ?fy ?tx ?ty))
    (true (cell ?x ?y ?v))
    (or (distinct ?x ?fx) (distinct ?y ?fy))
```

As the analysis of partiality of a game is superficial, it will not be able to see that the effects of the move is dependent on the player name. This analysis will simply see that the next relations will look similar for both players, and then incorrectly assume that it is impartial. A better analysis of partiality might check explicitly if the next relation uses the player name variable, and if so, mark it as partial. However, this is merely a special case, and is not a general test of partiality.

### 6.2.3 Solving the game suite

These are the results when attempting to solve the subtraction games in the suite. In the tables, each heap in the game is presented on its own row. In each row, the size of the heap and its Nim sequence is presented.

Table 2: Nim sequence of game SG{1,2,5}

| Heap | Size | Nim sequence |
|---|---|---|
| 1 | 10 | 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1 |
| 2 | 11 | 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2 |
| 3 | 12 | 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0 |

Table 3: Nim sequence of game SG{1,5,7}

| Heap | Size | Nim sequence |
|---|---|---|
| 1 | 10 | 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0 |
| 2 | 11 | 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 |
| 3 | 12 | 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0 |

Table 4: Nim sequence of game SG{1,2,3} (all variations)

| Heap | Size | Nim sequence |
|---|---|---|
| 1 | 21 | 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1 |

### 6.2.4 Failed cases

There are a number of identified cases where these methods of solving subtracton games fail. Here we describe the incorrect output and discuss why

Table 5: Nim sequence of game SG{1,2}{1,5} single

| Heap | Size | Nim sequence |
|---|---|---|
| 1 | 7 | 0, 1, 2, 0, 1, 2, 0, 1 |
| 2 | 11 | 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 |

these games fail.

Table 6: Nim sequence of game SG{1,5,7} nonzero

| Heap | Size | Nim sequence |
|---|---|---|
| 1 | 10 | 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0 |
| 2 | 11 | 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 |
| 3 | 12 | 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0 |

In the game "SG{1,5,7} nonzero", the ending condition is that each heap has size two. Hence, the Nim sequences are incorrect, as they start at size zero. This problem is coupled with the difficulty of assigning integer numbers to the constants defined by the `succ` relations. Here, we assign the value zero to the first `succ` constant, and make the assumption that the heaps may assume any size. This will be a problem when the game designer chooses to define integer constants in addition to the range used by the heaps.

Table 7: Nim sequence of game SG{1,2}{1,5}

| Heap | Size | Nim sequence |
|---|---|---|
| 1 | 7 | 0, 1, 2, 0, 1, 2, 0, 1 |
| 2 | 11 | 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2 |

The game SG{1,2}{1,5} in table 7, has two heaps each of which have their own subtraction set. The first heap has the subtraction set $\{1, 2\}$, while the second heap has the subtraction set $\{1, 5\}$. However, the analysis fails as it assigns the subtraction set $\{1, 2, 5\}$ to both heaps. The reason this happens is because of how the rules for legal moves are written. See figure 7 for pseudo-code from the game SG{1,2}{1,5}.

In figure 7 it can be seen that the `legal` relation accepts a move in the form `(move ?idx ?v)`, where `?idx` is the index of the heap (1 or 2) and `?v` is the value of the heap. The decision of whether this is a legal move is delegated to the relation `pmove`, which decides that it is legal if `?idx` is 1 and `?v` is the current value subtracted by one or two, or, if `?idx` is 2 and `?v` is the current value subtracted by one or five. The problem is that the software, when it builds its subtraction graph, searches for the conditions of a relation matching `(pmove ?idx ?v)`. Since it does not know the value of any of these two arguments, it matches every constant to these arguments.

Hence, for the legal relation all four different `pmove` implementations are matched, regardless of heap, and so incorrectly assigns the subtraction set $\{1, 2, 5\}$ to all players.

This case could be accounted for, if a more careful analysis was made. One could start the analysis by looking at each `next` relation and thereby being able to assign a value to `?idx`.

```
(<= (legal ?player (move ?idx ?v))
    (pmove ?idx ?v)
    (open)
    (true (control ?player)))

;; Subtraction set of heap 1
;; Subtraction by 1
(<= (pmove 1 ?v)
    (true (heap 1 ?oldv))
    (succ ?v ?oldv))

;; Subtraction by 2
(<= (pmove 1 ?v)
    (true (heap 1 ?oldv))
    (succ ?v ?a)
    (succ ?a ?oldv))

;; Subtraction set of heap 2
;; Subtraction by 1
(<= (pmove 2 ?v)
    (true (heap 2 ?oldv))
    (succ ?v ?oldv))

;; Subtraction by 5
(<= (pmove 2 ?v)
    (true (heap 2 ?oldv))
    (succ ?v ?a)
    (succ ?a ?b)
    (succ ?b ?c)
    (succ ?c ?d)
    (succ ?d ?oldv))

(<= (next (heap 1 ?v))
    (does ?player (move 1 ?v)))

(<= (next (heap 2 ?v))
    (does ?player (move 2 ?v)))
```

Figure 7: Pseudo code from game SG{1,2}{1,5}

# 7 Conclusions

Here the overall results of this master's project are presented. First the results and methods of the classification are discussed followed by results and methods of solving subtraction games. Finally, there is a general discussion on general game playing and the GDL framework.

## 7.1 Classification

The results for classification of each different property is presented here.

A general and significant problem when classifying combinatorial games is the multitude of ways in which to interpret and model certain aspects of a game. Often, combinatorial games can be interpreted in their "pure" theoretical form, or in a recreational way. This causes slight differences in how aspects of the games are written.

### 7.1.1 Normal and misère form

Classification of a game according to normal or misère forms is subject to the many ways in which one can determine how a game has ended. For instance, imagine the game of chess written in GDL. Chess is a combinatorial game in normal form, if you remove the possibility of a draw, but will most likely not be written as such. The terminal condition will most likely be that one player puts the other in check mate. However, a more "pure" combinatorial interpretation would be that the game ends one turn later, when one player cannot move because being in check mate implies that one does not have a valid move. Because of this, the methods presented for classification of the form of combinatorial games will most likely fail for any game of chess. Similar arguments can be made for practically any other game.

### 7.1.2 All small games

As GDL uses first order logic, the task of determining whether a game is "all small" is made rather easy. Making logical deductions based on the conditions of a move provides a simple yet effective way of determining whether an opposing player has a corresponding move.

This method is very specific to GDL and its first order logic representation of games.

### 7.1.3 Partial and impartial games

The method presented for determining whether a game is partial or impartial cannot be considered sufficient for any application. The solution is based solely on loose assumptions on syntactical properties of partial and impartial

games. However, determining partiality is no easy task as a great deal of interpretation is involved.

## 7.2 Solving subtraction games

As a result of the Game Description Languages unrestricted language structures, the success of trying to solve a general subtraction game is heavily dependent on how the game was written. For instance, a heap may be represented in a number of ways in the game code. This master project's solution only accounts for two of those ways, with the motivation that these two seemingly are the most obvious.

The use of arithmetic is naturally an important part of any subtraction game. If the methods of arithmetic cannot be properly analysed, then the remaining parts of the subtraction game solution will fail. In this light, it can be argued that the absence of arithmetical operations in GDL is a serious flaw.

The tests performed on this solution and the difficulties pointed out clearly show that these methods cannot be used to provide a general solution to the problem. Most parts of the solution require that the game designer use highly specific syntactical and semantic structures when writing the game code. In the absence of a more specialised general game playing framework, however, this must be expected.

The analysis above does not take into account the terminal condition placed on the game. In a subtraction game, the terminal condition will most commonly be that all heaps are empty.

## 7.3 General game playing and GDL

The use of the Game Description Language framework for combinatorial games results in a number of difficulties. A major difficulty rests in that GDL can be viewed as being designed for software that maintains a set of true statements about a game and, in each turn, chooses a move based on these statements. This does not work very well with the game theoretic view assumed in this master's project. A game theoretic view defines a set of rules applicable to a any valid game position, while the Game Description Language can only define rules for a specific game position. Translating from GDL to the game theoretic view appears to be difficult.

## 7.4 Future work

What follows are suggestions for further work.

### 7.4.1 Classification of cyclic games

An important classification of games not used here is cyclic or acyclic games. A cyclic game is a game where a specific game state may occur on several occasions during the game. This opens up the possibility of infinite play. Combinatorial games are by definition acyclic, that is, one can never reach the same game state twice in a game.

# References

[1] E. Berlekamp, J. Conway, and R. Guy. *Winning ways for your Mathematical Plays*, volume 1. A. K. Peters, 2nd, edition, 2004.

[2] E. Berlekamp, J. Conway, and R. Guy. *Winning ways for your Mathematical Plays*, volume 2. A. K. Peters, 2nd, edition, 2004.

[3] J. Conway. *On Numbers And Games*. Academic Press Inc., 1976.

[4] N. Love, T. Hinrichs, and M. Genesereth. *General Game Playing: Game Description Language Specification*, April 2006. http://games.stanford.edu/gdl_spec.pdf.

[5] B. Pell. Metagame: A new challenge for games and learning. *Internet*, 1992. http://www.barneypell.com/papers/metagame-olympiadUCAM-CL-TR-276.pdf.

[6] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3), March 1985. Website viewed 21 June, 2006, http://www.research.ibm.com/massive/tdl.html.

# A Game rules

## A.1 Clobber

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role white)
(role black)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Initialise board.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (cell 1 1 white))
(init (cell 1 2 black))
(init (cell 1 3 white))
(init (cell 1 4 black))

(init (cell 2 1 black))
(init (cell 2 2 white))
(init (cell 2 3 black))
(init (cell 2 4 white))

(init (cell 3 1 white))
(init (cell 3 2 black))
(init (cell 3 3 white))
(init (cell 3 4 black))

(init (cell 4 1 black))
(init (cell 4 2 white))
(init (cell 4 3 black))
(init (cell 4 4 white))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; White player begins.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control white))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal ?p (move ?fx ?fy ?tx ?ty))
    (canmove ?p ?fx ?fy ?tx ?ty))

(<= (legal ?p noop)
    (true (control ?q))
    (distinct ?p ?q))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (cell ?fx ?fy blank))
    (does ?p (move ?fx ?fy ?tx ?ty)))

(<= (next (cell ?tx ?ty ?p))
    (does ?p (move ?fx ?fy ?tx ?ty)))

(<= (next (cell ?x ?y ?v))
    (does ?p (move ?fx ?fy ?tx ?ty))
    (true (cell ?x ?y ?v))
    (or (distinct ?x ?fx) (distinct ?y ?fy))
    (or (distinct ?x ?tx) (distinct ?y ?ty)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control white))
    (true (control black)))

(<= (next (control black))
    (true (control white)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal white 0)
(true (control black)))

(<= (goal black 0)
(true (control white)))

(<= (goal white 100)
(true (control white)))

(<= (goal black 100)
(true (control black)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (terminal)
    (not (canmove ?p ?fx ?fy ?tx ?ty)))

(<= (canmove ?p ?fx ?fy ?tx ?ty)
    (true (control ?p))
    (adjacent ?fx ?fy ?tx ?ty)
    (true (cell ?fx ?fy ?p))
    (true (cell ?tx ?ty ?q))
    (distinct ?q ?p)
    (distinct ?q blank))

(<= (adjacent ?x1 ?y ?x2 ?y)
    (adjacent ?x1 ?x2))

(<= (adjacent ?x1 ?y1 ?x2 ?y2)
    (adjacent ?y1 ?x1 ?y2 ?x2))

(adjacent 1 2)
(adjacent 2 3)
(adjacent 3 4)
(<= (adjacent ?x ?y)
    (adjacent ?y ?x))
```

## A.2 Domineering

```
(role updown)
(role leftright)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3x4 board
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 1 4 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 2 4 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (cell 3 4 b))
(init (control updown))
(succ 1 2)
(succ 2 3)
(succ 3 4)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Checks if the board is marked blank,
;; at coordinates (x,y) and (x,y-1)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (freedown ?x ?y)
    (true (succ ?w ?y))
    (true (cell ?x ?y b))
    (true (cell ?x ?w b)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Checks if the board is marked blank,
;; at coordinates (x,y) and (x-1,y)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (freeside ?x ?y)
    (true (succ ?v ?x))
    (true (cell ?x ?y b))
    (true (cell ?v ?y b)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Moving at (x,y) is legal for updown,
;; if (x,y) and (x,y-1) are both blank.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal updown (move ?x ?y))
    (freedown ?x ?y)
    (true (control updown)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Moving at (x,y) is legal for leftright,
;; if (x,y) and (x-1,y) are both blank.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal leftright (move ?x ?y))
    (true (freeside ?x ?y))
    (true (control leftright)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Noop moves
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal leftright noop)
    (true (control updown)))

(<= (legal updown noop)
    (true (control leftright)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Mark appropriate cells as set,
;; when a player makes his move.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (cell ?x ?y s))
    (does updown (move ?x ?y)))

(<= (next (cell ?x ?w s))
    (does updown (move ?x ?y))
    (true (succ ?w ?y)))

(<= (next (cell ?x ?y s))
    (does leftright (move ?x ?y)))

(<= (next (cell ?v ?y s))
    (does leftright (move ?x ?y))
    (true (succ ?v ?x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; State that untouched cells retain
;; their mark
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (cell ?x ?y b))
    (true (cell ?x ?y b))
    (does updown (move ?w ?v))
    (distinct ?x ?w)
    (distinct ?y ?v)
    (succ ?m ?v)
    (distinct ?m ?y))

(<= (next (cell ?x ?y b))
    (true (cell ?x ?y b))
    (does leftright (move ?w ?v))
    (distinct ?x ?w)
    (distinct ?y ?v)
    (succ ?m ?w)
    (distinct ?m ?x))

(<= (next (cell ?x ?y s))
    (true (cell ?x ?y s)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Alternating turns
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (control leftright))
    (true (control updown)))

(<= (next (control updown))
    (true (control leftright)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Goal and terminal relations
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (goal updown 100)
    (true (control leftright)))

(<= (goal updown 0)
    (true (control updown)))

(<= (goal leftright 100)
    (true (control updown)))

(<= (goal leftright 0)
    (true (control leftright)))

(<= terminal
    (true (control updown))
    (not (freedown 1 1))
    (not (freedown 2 1))
    (not (freedown 1 2))
    (not (freedown 2 2))
    (not (freedown 1 3))
    (not (freedown 2 3))
    (not (freedown 1 4))
    (not (freedown 2 4)))

(<= terminal
    (true (control leftright))
    (not (freeside 1 2))
    (not (freeside 1 3))
    (not (freeside 1 4))
    (not (freeside 2 2))
    (not (freeside 2 3))
    (not (freeside 2 4))
    (not (freeside 3 2))
    (not (freeside 3 3))
    (not (freeside 3 4)))
```

## A.3 Kayles

```
(role left)
(role right)
(init (skittle 1 up))
(init (skittle 2 up))
(init (skittle 3 up))
(init (skittle 4 up))
(init (skittle 5 up))
(init (skittle 6 up))
(init (skittle 7 up))
(init (control left))
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Check if you can remove a single
;; skittle at coordinate a,
;; or two skittles at coordinate a
;; and a+1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (one ?a)
    (skittle ?a up))

(<= (two ?a)
    (skittle ?a up)
    (succ ?a ?b)
    (skittle ?b up))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Legal moves
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (single ?a))
    (true (one ?a))
    (true (control ?player)))

(<= (legal ?player (double ?a))
    (true (two ?a))
    (true (control ?player)))

(<= (legal left noop)
    (true (control right)))

(<= (legal right noop)
    (true (control left)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Removing skittles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (skittle ?a down))
    (does ?player (single ?a)))

(<= (next (skittle ?a down))
    (does ?player (double ?a)))

(<= (next (skittle ?b down))
    (does ?player (double ?a))
    (succ ?a ?b))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Maintaining untouched skittles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (skittle ?a up))
    (true (skittle ?a up))
    (does ?player (single ?b))
    (distinct ?a ?b))

(<= (next (skittle ?a up))
    (true (skittle ?a up))
    (does ?player (double ?b))
    (succ ?b ?c)
    (distinct ?a ?b)
    (distinct ?a ?c))

(<= (next (skittle ?a down))
    (true (skittle ?a down)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Alternating turns
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control left))
    (true (control right)))

(<= (next (control right))
    (true (control left)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Goal and terminal relations
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (goal left 100)
    (true (control right)))

(<= (goal left 0)
    (true (control left)))

(<= (goal right 100)
    (true (control left)))

(<= (goal right 0)
    (true (control right)))

(<= terminal
    (true (skittles 1 down))
    (true (skittles 2 down))
    (true (skittles 3 down))
    (true (skittles 4 down))
    (true (skittles 5 down))
    (true (skittles 6 down))
    (true (skittles 7 down)))
```

## A.4 Turning turtles

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role left)
(role right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Initialise board.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (turtle 1 feet))
(init (turtle 2 feet))
(init (turtle 3 feet))
(init (turtle 4 feet))
(init (turtle 5 feet))
(init (turtle 6 feet))
(init (turtle 7 feet))
(init (turtle 8 feet))
(init (turtle 9 feet))
(init (turtle 10 feet))
(init (turtle 11 feet))
(init (turtle 12 feet))
(init (turtle 13 feet))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal ?p (turn ?x))
    (true (control ?p))
    (true (turtle ?x feet)))

(<= (legal ?p (turn ?x ?y))
    (true (control ?p))
    (true (turtle ?x feet))
    (succ ?y ?x))

(<= (legal ?p noop)
    (true (control ?q))
    (distinct ?p ?q))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (turtle ?x back))
    (or (does ?p (turn ?x)) (does ?p (turn ?x ?y))))

(<= (next (turtle ?y back))
    (does ?p (turn ?x ?y))
    (true (turtle ?y feet)))

(<= (next (turtle ?y feet))
    (does ?p (turn ?x ?y))
    (true (turtle ?y back)))

(<= (next (turtle ?z ?v))
    (or (does ?p (turn ?x)) (does ?p (turn ?x ?y)))
    (true (turtle ?z ?v))
    (distinct ?x ?z)
    (distinct ?y ?z))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control left))
    (true (control right)))

(<= (next (control right))
    (true (control left)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal ?p 100)
    (true (control ?q))
    (distinct ?p ?q))

(<= (goal ?p 0)
    (true (control ?p)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not (true (turtle ?x feet))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Arithmetic.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
(succ 12 13)
```

## A.5 Fox and geese

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role fox)
(role goose)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Initialise board.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (cell 1 2 goose))
(init (cell 1 4 goose))
(init (cell 1 6 goose))
(init (cell 1 8 goose))

(init (cell 2 1 empty))
(init (cell 2 3 empty))
(init (cell 2 5 empty))
(init (cell 2 7 empty))

(init (cell 3 2 empty))
(init (cell 3 4 empty))
(init (cell 3 6 empty))
(init (cell 3 8 empty))

(init (cell 4 1 empty))
(init (cell 4 3 empty))
(init (cell 4 5 empty))
(init (cell 4 7 empty))

(init (cell 5 2 empty))
(init (cell 5 4 empty))
(init (cell 5 6 empty))
(init (cell 5 8 empty))

(init (cell 6 1 empty))
(init (cell 6 3 empty))
(init (cell 6 5 empty))
(init (cell 6 7 empty))

(init (cell 7 2 empty))
(init (cell 7 4 empty))
(init (cell 7 6 empty))
(init (cell 7 8 empty))

(init (cell 8 1 empty))
(init (cell 8 3 empty))
(init (cell 8 5 fox))
(init (cell 8 7 empty))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Fox begins.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control fox))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal fox (move ?fx ?fy ?tx ?ty))
    (foxmove ?fx ?fy ?tx ?ty))

(<= (legal goose (move ?fx ?fy ?tx ?ty))
    (goosemove ?fx ?fy ?tx ?ty))

(<= (legal ?p noop)
    (true (control ?q))
    (distinct ?p ?q))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (cell ?fx ?fy empty))
    (does ?p (move ?fx ?fy ?tx ?ty)))

(<= (next (cell ?tx ?ty ?p))
    (does ?p (move ?fx ?fy ?tx ?ty)))

(<= (next (cell ?x ?y ?v))
    (does ?p (move ?fx ?fy ?tx ?ty))
    (true (cell ?x ?y ?v))
    (or (distinct ?x ?fx) (distinct ?y ?fy))
    (or (distinct ?x ?tx) (distinct ?y ?ty)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control fox))
(true (control goose)))

(<= (next (control goose))
(true (control fox)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal ?p 100)
    (true (control ?q))
    (distinct ?p ?q))

(<= (goal ?p 0)
    (true (control ?p)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not (foxmove ?fx ?fy ?tx ?ty))
    (not (goosemove ?fx ?fy ?tx ?ty)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Legal move helpers.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (foxmove ?fx ?fy ?tx ?ty)
    (true (control fox))
    (true (cell ?fx ?fy fox))
    (true (cell ?tx ?ty empty))
    (adjacent ?fx ?tx)
    (adjacent ?fy ?ty))

(<= (goosemove ?fx ?fy ?tx ?ty)
    (true (control goose))
    (true (cell ?fx ?fy goose))
    (true (cell ?tx ?ty empty))
    (succ ?fx ?tx)
    (adjacent ?fy ?ty))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Arithmetic.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
(succ 12 13)
(succ 13 14)
(succ 14 15)
(succ 15 16)
(succ 16 17)
(succ 17 18)
(succ 18 19)
(succ 19 20)
(succ 20 21)
(succ 21 22)
(succ 22 23)
(succ 23 24)
(succ 24 25)

(<= (adjacent ?x ?y)
    (or (succ ?x ?y) (succ ?y ?x)))
```

## A.6 SG{1,2,3}(normal form)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    (open))

(<= (legal Right noop)
    (not (true (control Right)))
    (open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not (open))
    (not (true (control Left))))

(<= (goal Left 0)
    (not (open))
    (true (control Left)))

(<= (goal Right 100)
    (not (open))
    (not (true (control Right))))

(<= (goal Right 0)
    (not (open))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not (open)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Start value of heap.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (value 21))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (open)
    (not (true (value 0))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal ?player (move ?x))
    (true (control ?player))
    (open)
    (pmove ?x))

(<= (pmove ?x)
(true (value ?y))
(succ ?x ?y))

(<= (pmove ?x)
(true (value ?y))
(succ ?x ?a)
(succ ?a ?y))

(<= (pmove ?x)
(true (value ?y))
(succ ?x ?a)
(succ ?a ?b)
(succ ?b ?y))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (value ?x))
(does ?player (move ?x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define arithmetic.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
(succ 12 13)
(succ 13 14)
(succ 14 15)
(succ 15 16)
(succ 16 17)
(succ 17 18)
(succ 18 19)
(succ 19 20)
(succ 20 21)
```

## A.7 SG{1,2,3}(misère form)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    (open))
(<= (legal Right noop)
    (not (true (control Right)))
    (open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Misere form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not (open))
    (true (control Left)))

(<= (goal Left 0)
    (not (open))
    (not (true (control Left))))

(<= (goal Right 100)
    (not (open))
    (true (control Right)))

(<= (goal Right 0)
    (not (open))
    (not (true (control Right))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not (open)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Start value of heap.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (value 21))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (open)
    (not (true (value 0))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (move ?x))
    (true (control ?player))
    (open)
    (pmove ?x))

(<= (pmove ?x)
(true (value ?y))
(succ ?x ?y))

(<= (pmove ?x)
(true (value ?y))
(succ ?x ?a)
(succ ?a ?y))

(<= (pmove ?x)
(true (value ?y))
(succ ?x ?a)
(succ ?a ?b)
(succ ?b ?y))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (value ?x))
(does ?player (move ?x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define arithmetic.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
(succ 12 13)
(succ 13 14)
(succ 14 15)
(succ 15 16)
(succ 16 17)
(succ 17 18)
(succ 18 19)
(succ 19 20)
(succ 20 21)
```

# A.8 SG{1,2,5}

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    open)

(<= (legal Right noop)
    (not (true (control Right)))
    open)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not open)
    (not (true (control Left))))

(<= (goal Left 0)
    (not open)
    (true (control Left)))

(<= (goal Right 100)
    (not open)
    (not (true (control Right))))

(<= (goal Right 0)
    (not open)
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define initialised relations here.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (heap 1 10))
(init (heap 2 11))
(init (heap 3 12))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (open) (not (and (true (heap 1 0))
                     (true (heap 2 0))
                     (true (heap 3 0)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (move ?idx ?v))
    (pmove ?idx ?v)
(open)
(true (control ?player)))

;; Subtraction by 1
(<= (pmove ?idx ?v)
(true (heap ?idx ?oldv))
(succ ?v ?oldv))

;; Subtraction by 2
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?oldv))

;; Subtraction by 5
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?oldv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (heap ?idx ?v))
    (does ?player (move ?idx ?v)))

;; Define arithmetic
(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
```

## A.9 SG{1-5-7}

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    open)

(<= (legal Right noop)
    (not (true (control Right)))
    open)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not open)
    (not (true (control Left))))

(<= (goal Left 0)
    (not open)
    (true (control Left)))

(<= (goal Right 100)
    (not open)
    (not (true (control Right))))

(<= (goal Right 0)
    (not open)
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Misere form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(<= (goal Left 100)
;    (not open)
;    (true (control Left)))
;
;(<= (goal Left 0)
;    (not open)
;    (not (true (control Left))))
;
;(<= (goal Right 100)
;    (not open)
;    (true (control Right)))
;
;(<= (goal Right 0)
;    (not open)
;    (not (true (control Right))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define initialised relations here.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (heap 1 10))
(init (heap 2 11))
(init (heap 3 12))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (open) (not (and (true (heap 1 0)) (true (heap 2 0)) (true (heap 3 0)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (move ?idx ?v))
    (pmove ?idx ?v)
(open)
(true (control ?player)))

;; Subtraction by 1
(<= (pmove ?idx ?v)
(true (heap ?idx ?oldv))
(succ ?v ?oldv))

;; Subtraction by 5
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?oldv))

;; Subtraction by 7
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?e)
(succ ?e ?f)
(succ ?f ?oldv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (heap ?idx ?v))
    (does ?player (move ?idx ?v)))

;; Define arithmetic
(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
```

## A.10 SG{3-5-7}

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternerating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    open)

(<= (legal Right noop)
    (not (true (control Right)))
    open)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not open)
    (not (true (control Left))))

(<= (goal Left 0)
    (not open)
    (true (control Left)))

(<= (goal Right 100)
    (not open)
    (not (true (control Right))))

(<= (goal Right 0)
    (not open)
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Misere form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(<= (goal Left 100)
;    (not open)
;    (true (control Left)))
;
;(<= (goal Left 0)
;    (not open)
;    (not (true (control Left))))
;
;(<= (goal Right 100)
;    (not open)
;    (true (control Right)))
;
;(<= (goal Right 0)
;    (not open)
;    (not (true (control Right))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define initialised relations here.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (heap 1 10))
(init (heap 2 11))
(init (heap 3 12))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (open) (not (and (true (heap 1 0)) (true (heap 2 0)) (true (heap 3 0)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (heap ?idx ?v))
    (pmove ?idx ?v)
(open)
(true (control ?player)))

;; Subtraction by 3
(<= (pmove ?idx ?v)
(true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?oldv))

;; Subtraction by 5
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?oldv))

;; Subtraction by 7
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?e)
(succ ?e ?f)
(succ ?f ?oldv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (heap ?idx ?v))
    (does ?player (move ?idx ?v)))

;; Define arithmetic
(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
```

# A.11 SG{1,2}{1,5} single

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    open)

(<= (legal Right noop)
    (not (true (control Right)))
    open)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not open)
    (not (true (control Left))))

(<= (goal Left 0)
    (not open)
    (true (control Left)))

(<= (goal Right 100)
    (not open)
    (not (true (control Right))))

(<= (goal Right 0)
    (not open)
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Misere form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(<= (goal Left 100)
;    (not open)
;    (true (control Left)))
;
;(<= (goal Left 0)
;    (not open)
;    (not (true (control Left))))
;
;(<= (goal Right 100)
;    (not open)
;    (true (control Right)))
;
;(<= (goal Right 0)
;    (not open)
;    (not (true (control Right))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define initialised relations here.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (heap1 7))
(init (heap2 11))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (open) (not (and (true (heap1 0)) (true (heap2 0)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (move1 ?v))
    (pmove1 ?v)
(open)
(true (control ?player)))

(<= (legal ?player (move2 ?v))
    (pmove2 ?v)
(open)
(true (control ?player)))

;; Subtraction set of heap 1
;; Subtraction by 1
(<= (pmove1 ?v)
(true (heap1 ?oldv))
(succ ?v ?oldv))

;; Subtraction by 2
(<= (pmove1 ?v)
(true (heap1 ?oldv))
(succ ?v ?a)
(succ ?a ?oldv))

;; Subtraction set of heap 2
;; Subtraction by 1
(<= (pmove2 ?v)
    (true (heap2 ?oldv))
(succ ?v ?oldv))

;; Subtraction by 5
(<= (pmove2 ?v)
    (true (heap2 ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?oldv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (heap1 ?v))
    (does ?player (move1 ?v)))

(<= (next (heap2 ?v))
    (does ?player (move2 ?v)))

;; Define arithmetic
(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
```

# A.12 SG{1,2}{1,5}

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternerating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    open)

(<= (legal Right noop)
    (not (true (control Right)))
    open)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not open)
    (not (true (control Left))))

(<= (goal Left 0)
    (not open)
    (true (control Left)))

(<= (goal Right 100)
    (not open)
    (not (true (control Right))))

(<= (goal Right 0)
    (not open)
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define initialised relations here.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (heap 1 7))
(init (heap 2 11))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (open) (not (and (true (heap 1 0))
                     (true (heap 2 0)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (move ?idx ?v))
    (pmove ?idx ?v)
(open)
(true (control ?player)))

;; Subtraction set of heap 1
;; Subtraction by 1
(<= (pmove 1 ?v)
(true (heap 1 ?oldv))
(succ ?v ?oldv))

;; Subtraction by 2
(<= (pmove 1 ?)
(true (heap 1 ?oldv))
(succ ?v ?a)
(succ ?a ?oldv))

;; Subtraction set of heap 2
;; Subtraction by 1
(<= (pmove 2 ?v)
    (true (heap 2 ?oldv))
(succ ?v ?oldv))

;; Subtraction by 5
(<= (pmove 2 ?v)
    (true (heap 2 ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?oldv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (heap ?idx ?v))
    (does ?player (move ?idx ?v)))

;; Define arithmetic
(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
```

## A.13 SG{1,5,7} nonzero

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Two players
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(role Left)
(role Right)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Left player begins. Arbitrarily chosen.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(init (control Left))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Alternerating turns.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (next (control Right))
    (true (control Left)))

(<= (next (control Left))
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Noop moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (legal Left noop)
    (not (true (control Left)))
    open)

(<= (legal Right noop)
    (not (true (control Right)))
    open)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Goal conditions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Normal form
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= (goal Left 100)
    (not open)
    (not (true (control Left))))

(<= (goal Left 0)
    (not open)
    (true (control Left)))

(<= (goal Right 100)
    (not open)
    (not (true (control Right))))

(<= (goal Right 0)
    (not open)
    (true (control Right)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Terminal condition. End of game.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(<= terminal
    (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; Define initialised relations here.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (heap 1 10))
(init (heap 2 11))
(init (heap 3 12))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define open here, that is
;;; the game is not over if open is true.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (open) (not (true (heap 1 2))
                (true (heap 2 2))
                (true (heap 3 2))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define legal moves.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (move ?idx ?v))
    (pmove ?idx ?v)
(open)
(true (control ?player)))

;; Subtraction by 1
(<= (pmove ?idx ?v)
(true (heap ?idx ?oldv))
(succ ?v ?oldv))

;; Subtraction by 5
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?oldv))

;; Subtraction by 7
(<= (pmove ?idx ?v)
    (true (heap ?idx ?oldv))
(succ ?v ?a)
(succ ?a ?b)
(succ ?b ?c)
(succ ?c ?d)
(succ ?d ?e)
(succ ?e ?f)
(succ ?f ?oldv))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define state of game in the next
;;; turn.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (next (heap ?idx ?v))
    (does ?player (move ?idx ?v)))

;; Define arithmetic
(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
```

# B Game Description Language grammar

This is the grammar used for parsing GDL code, written in ANTLR syntax. The grammar is stripped of all implementation specific code.

```
header {
package spela.losa;
import spela.losa.util.Logger;
import spela.losa.grammar.*;
}
/** Grammar for the Game Description Language
 *
 * @author Erik Edin
 */

class GDLParser extends Parser;

options {
  k=5;
}

gameDescription :
      (WS)?
      (
        ( role | init | leadsto | relation | legal )
        (WS)?
      )*;

role : LPAR "role" WS id RPAR ;

init : LPAR "init" WS relation  (WS)? RPAR ;

relation : ( LPAR n1:NAME  (WS argument)* RPAR | n2:NAME ) ;

argument : ( var:VARIABLE | relation ) ;

legal : LPAR "legal" WS id WS move (WS)? RPAR ;

move : LPAR n1:NAME  (WS id)+ RPAR | n2:NAME | var:VARIABLE ;

trueSentence : LPAR "true" WS relation (WS)? RPAR ;

next : LPAR "next" WS relation (WS)? RPAR ;
```

```
sentence :
      (
        trueSentence
        |
        relation
        |
        does
        |
        role
        |
        negation
        |
        or
        |
        and
        |
        goal
      )
    ;

negation : LPAR "not" WS sentence (WS)? RPAR ;

or : LPAR "or" WS (sentence (WS)?)+ RPAR ;

and : LPAR "and" WS (sentence (WS)?)+ RPAR ;

leadsto :
      (
        LPAR "<=" WS legal (WS)? (sentence (WS)?)* RPAR
        |
        LPAR "<=" WS next (WS) (sentence (WS)?)* RPAR
        |
        LPAR "<=" WS goal WS (sentence (WS)?)* RPAR
        |
        LPAR "<=" WS relation (WS sentence)* RPAR
      )
    ;

does : ( LPAR "does" WS id WS move (WS)? RPAR) ;

goal : LPAR "goal" WS id WS value:NAME (WS)? RPAR ;

id : ( n1:NAME | v1:VARIABLE ) ;
```

```
class GDLLexer extends Lexer;

options {
  k=6;
}

LPAR
    :
      '('
    ;

RPAR
    :
      ')'
    ;

NAME
    :
      (~(' '|'\n'|'\t'|'\r'|'('|')'|';'|'?'))+
    ;

VARIABLE
    :
      '?' NAME
    ;

protected
COMMENT
    :
       ';' (~('\r'|'\n'))* ('\r'|'\n')
    ;

WS
    :
      ('\n' | '\t' | ' ' | '\r' | COMMENT)+
    ;
```

TRITA-CSC-E 2007:048
ISRN-KTH/CSC/E--07/048--SE
ISSN-1653-5715

www.kth.se