CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Language for Board Games

Development of an Embedded Domain-Specific Language for Describing Board Games

Bachelor's thesis in Computer science and engineering

Edvin Alestig
Joel Ericson
Erik Eriksson
Lukas Schiavone
Filip Torphage
Joakim Tubring

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Bachelor's thesis 2022

A Language for Board Games

Development of an Embedded Domain-Specific Language
for Describing Board Games

Edvin Alestig
Joel Ericson
Erik Eriksson
Lukas Schiavone
Filip Torphage
Joakim Tubring

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A Language for Board Games
Development of an Embedded Domain-Specific Language
for Describing Board Games

Edvin Alestig Joel Ericson Erik Eriksson Lukas Schiavone Filip Torphage
Joakim Tubring

Supervisor: Robin Adams, Department of Computer Science and Engineering
Examiner: Alex Gerdes, Department of Computer Science and Engineering

Bachelor's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

A Language for Board Games
Development of an Embedded Domain-Specific Language
for Describing Board Games

Edvin Alestig, Joel Ericson, Erik Eriksson, Lukas Schiavone,
Filip Torphage, Joakim Tubring

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In recent years board games have increasingly found themselves in the digital medium. One way to enable easier creation of digital board games is to create a domain-specific language (DSL) for that purpose. This thesis details the process of developing an embedded DSL for describing board games with Haskell as its host language. The goal is for users to be able to develop a large number of board games using the language. How the DSL was created is explained. Also included is a detailed guide on how to create board games using the DSL as well as a list of all tools available to the user of the language. Alongside the guide are examples of different board games written in the language. After the results are presented a description is given on the inner workings of how a game is run based on its specifications. Thereafter the results of the project are thoroughly discussed and guidelines are given on how the DSL could be improved and expanded upon in the future.

Keywords: DSL, domain-specific languages, Haskell, functional programming, board games, game theory

Ett språk för brädspel
Utveckling av ett inbäddat domänspecifikt språk för att beskriva brädspel

Edvin Alestig, Joel Ericson, Erik Eriksson, Lukas Schiavone,
Filip Torphage, Joakim Tubring

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola och Göteborgs Universitet

Sammandrag

På senare tid har brädspel alltmer befunnit sig i det digitala mediet. Ett sätt att möjliggöra skapandet av digitala brädspel är att skapa ett domänspecifikt språk (DSL) för det syftet. Den här avhandlingen beskriver processen för att utveckla ett DSL i Haskell för att beskriva brädspel samt bakgrundsteori om DSL:er och spelteori. Målet med språket är att möjliggöra skapandet av ett stort antal brädspel. Hur DSL:en skapades förklaras. Inkluderat är också en detaljerad guide om hur man skapar brädspel med hjälp av språket samt en lista över alla verktyg som är tillgängliga för användaren av språket. Vid sidan av guiden finns exempel på olika brädspel skrivna i språket. Efter att resultaten presenterats ges en beskrivning av hur ett brädspel körs internt, baserat på dess specifikationer. Därefter diskuteras projektets resultat och riktlinjer ges för hur språket kan förbättras och utökas i framtiden.

Nyckelord: DSL, domänspecifika språk, Haskell, funktionell programmering, brädspel, spelteori

Acknowledgements

We would like to give a big thank you to our supervisor Robin Adams. He has helped us a lot with the technical aspects of the project as well as with this report. We would also like to thank our examiner Alex Gerdes for his help with the planning report.

Edvin Alestig, Joel Ericson, Erik Eriksson, Lukas Schiavone, Filip Torphage, Joakim Tubring
Gothenburg, May 2022

Contents

List of Figures

List of Tables

Nomenclature

Cooperative game	Games where players are encouraged to work together to maximise their own gain.
Deep embedded DSL	A type of embedding where the syntax is stored in a data type and is interpreted by a run function.
DSL	Domain-Specific Language
eDSL	Embedded Domain-Specific Language
GPL	General Purpose Language
GUI	Graphical User Interface
Imperfect information	See perfect information.
IO	Input and output
Perfect information	A perfect information game is a game in which all players have the same information of the game. This may include the knowledge of previous moves, possible future moves and cards on a player's hand. Otherwise the game is called an imperfect game.
Run function	A function that interprets a DSL's semantics to produce a result.
Shallow embedded DSL	A type of embedding where all semantics are created using functions building a final value.
Simultaneous game	A game where players do actions without the knowledge of the other players' actions.
Zero-sum game	A game where the total gain of the players must be equal to the total expense of the players in the game.

1

Introduction

One popular form of entertainment and socialisation among groups is board games. Board games have been played throughout history and have been traced as far back as ancient Egyptian times. One of the oldest board games ever played is called Senet. Senet was played by higher-standing members of the ancient Egyptian society and evidence suggests that it was played as far back as 3100 BC [1]. In our modern day, board games have needed to adapt to a new medium, the digital one. Fortunately, board games are well fitted to this medium due to their structure and there are many digital board games already in existence.

Many board games share a common structure: A game begins in a particular state, followed by a number of turns. In each turn, a particular number of players make a particular amount of allowed moves until some condition for winning or a draw is met, all of which is based on the rules of the game. This common structure can be taken advantage of when creating digital versions of these games. One way this can be done is to create a Domain-Specific Language (DSL) for board games. A DSL is a language focused on helping you solve problems within one particular domain. A DSL can utilise the commonality of board games to make the creation of them easier, cutting down on implementation time and code complexity. The goal of this project is to create such a DSL, together with a set of sample games that are implemented with the DSL to showcase its capabilities.

A DSL like this has many uses, both for entertainment purposes but also for research and testing purposes. One could for instance write a program within its' framework that analyses a certain state of a game and then outputs the chances of each player winning. Done right, this would work for all games that can be described with the language without needing to make it specific to one game. This claim will be justified in section 4.4.3. Another reason as to why a DSL for describing board games would be useful is for the creation of brand new games. Using a DSL someone could quickly create a new game and then test it to see that it plays as intended.

There are multiple instances of computer languages made for creating board games. Perhaps the largest and most exhaustive one is a project called Ludii. It was created by a team at Maastricht University and they describe Ludii as a "general game system" [2], and is a DSL used for creating games (including but not limited to board games). They state that the game system can describe any traditional strategy game with over 700 of what is referred to as "concepts". The DSL created in this project

will have a smaller scope with fewer "concepts". This can mean that the DSL will be easier to learn and use but will consequently support a smaller set of games.

1.1 Problem

The problem that the group decided to solve for this project was to create a DSL that enables the user to easily create board games within the subset of board games that are supported. This problem can be broken down into sub-problems. To begin with, an accurate data type representation of a game needed to be made. This choice of the data type affected how the rest of the code was written, so it was important to have a solid base. Also, what operations are needed to be performed on the game in order to change its state according to the turns the players make and the rules applied.

The game type in itself needed to contain data types representing other components of the game including the board, rules, pieces, and players. How each of these components will be modelled and used by the users of the DSL are problems in and of themselves. The rules of board games vary in complexity and simply providing a collection of rules to the user does not suffice. Therefore, the DSL must provide the user with the ability to create complex rules using simple components and features the DSL provides. This is the only way to support a large number of games without having specific functionality for each game the DSL supports. For instance, let us say that a game rule should only be applied if a certain condition is met. Then there would need to be a way to be able to model conditional rules, similar to if statements in "ordinary" programming languages.

Additionally, there is a usability aspect to the DSL that needs to be considered. How is the user going to create a new board game? What must they write or specify to be able to model a game and run it? The goal is for the users to be able to model games as expressively and easily as possible. Achieving this was challenging.

1.2 Purpose

The purpose of this project was to develop an embedded DSL (eDSL) in Haskell for creating board games. The DSL should allow the user to model a large number of board games with varying complexity in an expressive way. The quality of the DSL will be measured by how complex the board games the user can implement are, as well as how easy it is to make them.

To aid in the fulfilment of the purpose, a set of goals were made for incrementally developing the DSL towards more and more complex games. These goals are as follows:

1. Create a DSL that can model the game tic-tac-toe. This game is characterised by players alternating turns placing static game pieces.

2. Develop the DSL further to allow the creation of Othello, which has intricate rules for what happens after each piece is placed.

3. Finally, model the game Chess using the DSL. This game has complex rules, specific to individual game pieces, and also has moving game pieces.

Note that these three games are only the "goal games" that are used as a guide for what kind of features should be worked on and when. More games were taken into consideration to get a broader view of more ideas, rules, and concepts during development.

1.3 Scope

Due to the sheer number of board games in existence certain limitations had to be imposed for this project to be completed in the given time frame. The first and most important limitation is that only board games where positions can be described by discrete two-dimensional coordinates on a rectangular board are supported. Other limitations regarding the structures of board games are the following:

- Rules that change during the course of the game (dynamic rules) are not supported.
- Only zero-sum games are supported.
- Cooperative games are not supported.
- Only perfect-information games are supported.
- Simultaneous/non-sequential games are not supported.

The terms mentioned above will be explained in section 2.3.

This project will not lay focus on a Graphical User Interface (GUI). The default user interface that is given to the user is text printing to, and user input from, the terminal. The time that would be spent on developing a GUI was instead spent on expanding the DSL to support a larger set of games.

1.4 Source Code

All of the source code can be found on the project's GitHub page:
`https://github.com/edvinalestig/kandidatarbete`

2

Background Theory

In the following sections, relevant information for understanding this project is presented. First, the programming language Haskell will be explained and its benefits for using it for development. Then, domain-specific languages and some of the different types of DSLs that exist are explained. Lastly, relevant game theory terms necessary to understand the scope will be explained.

2.1 Haskell

Functional programming languages have many unique features that can simplify the work of creating an eDSL. One well-known functional programming language is Haskell. It is a pure, statically typed, and lazy functional programming language that is based on lambda calculus [3].

Haskell offers a compact syntax that can be used to create an eDSL with less code than other common programming languages [4]. This is mainly because of its powerful type system, higher-order functions, and lazy semantics [5]. The type system in Haskell is an algebraic data type system, and according to Jeremy Gibbons, this is crucial for making a deep embedded DSL [6]. The different types of embedding, including deep embedding, will be explained in more detail in section 2.2. By being a programming language with lazy semantics Haskell also allows the use of infinite data types, opening up many options for writing a DSL. Higher-order functions are also built into the language, providing more options to minimise repetitive functions. Additionally, Gibbons also says that higher-order functions are more or less required for building a shallow embedded DSL. Having the features to write both deep and shallow embedding gives the option of choosing which one fits better.

Another advantage of Haskell is the fact that it is a pure programming language. By being pure, Haskell guarantees that its functions do not have any side effects without explicitly specifying them. If there were side effects, you could get unexpected behaviour by combining functions that change the state in ways that have not been tested or thought about. Therefore, writing in a language without side effects would be especially useful for a DSL where the users of it are free to use the language as they please.

2.2 Domain-Specific Languages

A domain-specific language is a language that is tailored to a specific domain [7]. Well-known DSLs include HTML for structuring web pages and CSS for stylising them. However, JavaScript, often used in combination with HTML and CSS, is not tied to a specific domain and is therefore not a DSL. Instead, it is a General-Purpose Language (GPL). This is because JavaScript has uses in many different domains, including web pages, servers, and other applications [8].

When working within a given domain, using a DSL can have many benefits over using a GPL. This is simply because DSLs are created with a single purpose in mind, as opposed to GPLs. Consequently, this means that DSLs can be more expressive and easier to use compared to using GPLs within the same domain [9]. One way this is achieved is by using abstraction as a way to remove *boilerplate code* [10], i.e. code that does not provide any new information to the reader but is still required to obtain the intended functionality. Their limited scope also means that they are, in theory, easier to learn and do not require you to necessarily have any knowledge of any general-purpose programming languages. This means that DSLs can be used by domain experts without a software development background. Of course, there are disadvantages to DSLs as well. For instance, having to learn a new language compared to using a GPL you are familiar with, or the difficulties of integrating code from multiple domains using different DSLs instead of using a single GPL.

2.2.1 Embedded and External Domain-Specific Languages

There are two kinds of DSLs, *embedded* (sometimes called internal) and *external*. An embedded DSL, or eDSL, is written in a GPL, which is often called the *host* GPL [7]. An eDSL can be constructed as a library for the host language. One advantage of creating an eDSL is that it can use the compiler or interpreter of its host language [4]. However, this means that the syntax is fixed to what is possible in the host language. This can create limitations in the expressiveness of the DSL.

External DSLs, on the other hand, use their own compiler or interpreter [11]. Naturally, this gives the creators of the DSL freedom to design the language as they please. They control the whole pipeline which includes parsing, syntax, as well as optimising. External DSLs can thus be powerful, but consequently difficult to create.

2.2.2 Types of embedding

There are different ways to embed a DSL in a host language [12]. The embedding can be either shallow, deep, or a combination of the two. The decision of whether to use a deep or shallow embedding depends on the purpose of the DSL. If there are few interpretations, a combination of the two can be preferable. This allows the easy addition of new constructs while also allowing multiple interpretations. Below, the difference between deep and shallow embedding is explained.

In a shallow embedding, the representation of the semantics of the language is done directly in the host language. The interpretation is directly described in the functions used. Figure 2.1 shows an example of a shallow embedding representing the operators "<" and ">". Later the same example will be used for a deep embedding. In this example, Haskell is used since a shallow embedding is directly connected to the host language used.

```
type Exp = Int -> Int -> Bool

eLt :: Exp
eLt e1 e2 = e1 < e2

eGt :: Exp
eGt e1 e2 = e1 > e2
```

Figure 2.1: An example of shallow embedding.

A deep embedded DSL defines a unique structure using a syntax tree where the order and types of arguments are specified. Then, the host language is used to parse, type check, and interpret the language. There are usually two major parts in a deep embedding. The syntax of the language, represented by composite data types, and the interpreter, also called the run function. Figure 2.2 shows the same example of the standard logical operators "<" and ">" but defined using deep embedding instead.

```
data Exp where
    Let :: Int -> Exp
    ELt :: Exp -> Exp -> Bool
    EGt :: Exp -> Exp -> Bool
```

Figure 2.2: An example of deep embedding.

Both `ELt` and `EGt` are non-associative. The next step is to parse the input. Evaluating the expression `1 < 3` would yield the following parse tree.

```
ELt (Let 1) (Let 3)
```

This is where the run function is used, it takes the syntax tree as the argument and uses the host language to compute the result. Figure 2.3 shows an example of a run function written in Haskell.

Now the host language can be used to perform the calculation. Another example of a DSL written both with a shallow and deep embedding can be seen in appendix A.

```
run :: Exp -> Bool
run (ELt (Let n1) (Let n2)) = n1 < n2
run (EGt (Let n1) (Let n2)) = n1 > n2
```

Figure 2.3: An example run function for the deep embedded example of standard logical operators.

2.3 Game Theory

Game theory is a branch of mathematics for finding mathematical models for processes where players are competing. These mathematical models are often divided into several different categories. Below are summaries of how some of these categories are defined in [13].

A *zero-sum game* is a game such that the total gain of the players must be equal to the total expense of the players in the game. This means that the gains of a set of players must mean an equal loss for another set of players. For example, in poker, the winnings a player makes are at the expense of others. A non-zero-sum game is a game where the total numbers of rewards and penalties may differ at the end of the game. For example, if the same game is played several times the end result might differ. It might be that in one game the points received by the players in the game might be larger than the penalties.

Zero-sum: rewards − penalties = 0
Non-zero-sum: rewards − penalties ≠ 0.

An example of this is the famous *Prisoner's Dilemma* where the total score (in this case, the amount of years imprisoned) can end with total scores that do not add to zero [14].

Cooperative games are games where players are encouraged to work together to maximise their own gain. In game theory, this is often framed as several players being able to form coalitions and, by knowing the outcomes of these coalitions, argue for a scenario that everyone can agree on that benefits the individual as much as possible. One common framing of this is games where all the players are a joint team that either wins or loses as a group. Sometimes this is framed as a zero-sum game where the players are a team and the game is framed as an opposing entity. In *Mysterium*, one player plays as the ghost and tries to instruct the other players to figure out who has murdered them [15]. This is a cooperative game since all the players are working towards achieving the same win state. Non-cooperative games are the opposite of this, where players are tasked to maximise their own gain without concerning themselves with benefiting others.

Another distinction people make in terms of classifying games is a split between whether the players have *perfect* or *imperfect information*. Games of perfect information mean that all players have information on all moves done by every other player and their inventories as well. In chess, the players know what moves can be

made at any given board state as well as every move that has been made, making it a game of perfect information. On the other hand, a game that hides information from players, such as poker, will be classified as a game of imperfect information. This includes all games where players have some kind of hidden inventory.

Simultaneous games are characterised by doing an action without the knowledge of the other players' actions. This is either achieved by having multiple players play simultaneously or having the players play sequentially but without knowledge of the other players' moves. This is in contrast to *sequential* games where the latter players have some knowledge about other players' earlier actions. This term is, again, well represented by chess as a sequential game, as players take their turns in order and get to see the opposing player's move before making their own, and the prisoner's dilemma as a simultaneous game, as both players make their move without the knowledge of the other player's decision.

3

Design

This chapter will go over everything there is to know about the implementation and usage of the DSL. First, this chapter will present how the DSL was developed. Then, a language description containing the information needed for using the DSL to model games, followed by a showcase of some of the example games created using the DSL. After that, a somewhat brief look at the implementation details of the DSL and how it works internally. Finally, a section about how testing was conducted on the code base to verify certain functionality.

3.1 Process

The language created was chosen to be an eDSL using Haskell as its host language. Choosing to write an embedded DSL meant that more time could go into developing the DSL rather than implementing basic functionality for compiling or interpreting the language. Selecting the host language as Haskell was not only for all the reasons brought up in section 2.1 but also because the group members were familiar with it.

Before beginning the development of the DSL, a few simple games were chosen for implementation to find the similarities between them. By studying the similarities between these games a base for the DSL could be formed based on these similarities. The four games that were chosen in the beginning were tic-tac-toe, Connect Four, Othello, and snakes and ladders. After these four games were finished development began on the DSL using their common structure as an outline.

3.1.1 Iterative Development

The development process was done in iterations. Each one consisted of developing the DSL to a state such that it supported the creation of a pre-decided game. As mentioned in section 1.2, these were tic-tac-toe, Othello, and chess. Again, more games were taken into consideration during the development process to get a larger variety of ideas, rules, and concepts for the DSL.

For the first iteration, the goal was to have enough functionality to implement tic-tac-toe, where pieces are static and the rules are simple. During the second part, there would have to be expansions to the DSL to enable the modelling of Othello. The original goal for the second iteration was the creation of Ludo. This

would include the support of moving pieces, multiple pieces on a tile and die rolls. However, due to changes in scope which excluded functionality necessary for being able to model Ludo, the second goal was changed from Ludo to Othello. The reasons for why the plan changed will be explained in more detail in section 4.1. Othello has similarities with tic-tac-toe, both games consist of two players taking turns placing pieces onto the game board. What makes Othello different is that the rules for placing pieces and the effects of placing a piece are much more complex. To accommodate for this, the rule system had to be reworked to allow for the user themselves to build up complex rules using smaller building blocks, instead of only using existing rules that were provided to them.

As the last goal of the project, the plan was to be able to model chess using the DSL. Chess has a complex set of rules with different pieces having different ways they move around the board, which includes rules based on whether or not a piece has been moved previously. A player cannot castle if they have already moved the king, for instance. The previous games all had static pieces so allowing for the movement of pieces and accompanying rules had to be implemented. Due to the work done to generalise and modularise the library during the second phase, not much effort was required to support the movement of pieces. In the end, however, not all rules for chess could be implemented. Castling and en passant did not get implemented, and to win you have to capture the king, meaning that our version of chess cannot end in a draw.

3.2 Language Description

This section describes everything a user of the DSL needs to know to use the library for creating board games on their own. First, a look at some important data types for understanding how to create games and rules, including the fundamental `Game` data type, followed by the types of rules as well as how they are defined. Then, the details of how exactly a game can be created, using the tools the DSL provides.

3.2.1 The `Game` Data Type

To use the DSL for creating board games the developer should use the `game` value, which is of type `Game`, and override the necessary fields for the particular game implementation. The game can then be played by passing it as an argument to the function `playGame :: Game -> IO ()`. Because the DSL is deep embedded, you, as a developer, can create your own functions for running the game. By doing so, you will have the freedom of choosing how the games will play. You can, for instance, do this to hook up a proper UI framework to the game or use it together with AIs.

The `Game` data type can be seen in figure 3.1. The fields you should consider overriding are `board`, `pieces`, `players`, `preTurnRules`, `rules`, `endConditions`, and `dispFunction`. How each of these fields works will be explained below.

The `Board` data type is defined as a list of lists of tiles, where a `Tile` is either `Empty`

```
data Game = Game
    {
        board          :: Board,
        pieces         :: [Piece],
        players        :: [Player],
        preTurnRules   :: [Rule],
        rules          :: [Rule],
        endConditions  :: [Rule],
        winner         :: Maybe Player,
        gameEnded      :: Bool,
        dispFunction   :: Game -> IO ()
    }
```

Figure 3.1: The implementation of the `Game` data type.

or contains a single piece, `PieceTile`. Tiles also contain a `Pos`, which is just a simple data type containing two integer values. How these types are defined in the code base can be seen in figure 3.2.

```
data Pos = Pos Int Int
    deriving (Eq, Show)

type Board = [[Tile]]

data Tile = PieceTile Piece Pos
          | Empty Pos
    deriving (Eq)
```

Figure 3.2: The implementation of the `Board` and `Tile` data types.

Players are described by a string that should be unique for that player. Pieces contain the player whose piece it is as well as a string of how it is represented, i.e. what kind of piece it is. The definition of `Piece` and `Player` can be seen in figure 3.3.

```
data Piece = Piece String Player
    deriving (Eq)

newtype Player = Player String
    deriving (Eq)
```

Figure 3.3: The implementation of the `Piece` and `Player` data types.

The last field in the `Game` data type is `dispFunction`. The default implementation has the `dispFunction` set as a "pretty print" function for displaying the game to the terminal. This function uses the piece identifier string directly at the place it

is located when displaying it, meaning that all pieces should consist of only one character for it to display properly.

3.2.2 Turns

A turn is specified to be an action on a single piece. An action is either placing a piece at a certain position or moving it from one position to another. Exactly how turns are defined can be seen in figure 3.4.

```
data Turn = Turn Piece Action
    deriving (Show)

data Action = Place Pos | Move Pos Pos
    deriving (Show)
```

Figure 3.4: The implementation of the `Turn` and `Action` data types.

3.2.3 Updates

The base type representing a change to the state of the game is called `Update`. Its definition can be seen in figure 3.5. For the game state to update, a turn must be played, hence why `Turn` is used. The parameter `t` is whatever type is updated, which, in the library will either be `Game` or `Turn`. You can chain multiple updates together using `COMBINE`.

```
data Update t = Update (Turn -> t -> t)
              | (Update t) `COMBINE` (Update t)
```

Figure 3.5: The implementation of the `Update` data type.

3.2.4 Conditions

Conditional rules will be used in almost all cases when specifying a board game. Using a non-conditional rule in a game would lead to it being applied every single turn, which might be useful in certain niche cases. Since conditional rules are so important it is equally if not more important to have the ability to construct complicated conditions from simpler conditions. The `Condition` data type is shown in figure 3.6 and shows the base `Condition` as well how more complex conditions can be constructed.

The first three operations `‘AND‘`, `‘OR‘` and `NOT` are simple boolean operators that do not need to be explained further. The last two operations, `All` and `Any` are a bit more complicated and work very similarly to each other. To use them one must specify a condition on the type `a` as well as a function that produces a list of `a`'s given a `Turn` and `Game` as inputs. The `All` condition will return `True` if all elements in the list fulfill the condition, `Any` will return `True` if at least one element fulfills the

```
data Condition a = Condition (a -> Game -> Bool)
                 | (Condition a) `AND` (Condition a)
                 | (Condition a) `OR`  (Condition a)
                 | NOT (Condition a)
                 | All (Condition a) (Turn -> Game -> [a])
                 | Any (Condition a) (Turn -> Game -> [a])
```

Figure 3.6: The implementation of the `Condition` data type.

condition. A use of `All` is shown in figure 3.7. For the condition `tilesBetweenAre` to return true, the condition `c` applied to the result of `tilesBetweenTwoCoords` must all be true.

```
tilesBetweenAre :: Condition Turn -> Condition Turn
tilesBetweenAre c = All c tilesBetweenTwoCoords
```

Figure 3.7: Using `All` to check that all tiles between two coordinates fulfill a condition.

A good example of a complex condition is the condition for if a move is equal to the move of a bishop in chess, seen in figure 3.8. Important to note is the naming convention here. Where a condition relates to will be included in the name, that is, either its origin or its destination of the move. In this case, it is `NOT allyDestination`, which means that the bishop cannot capture a piece of their own colour.

```
bishopMove :: Condition Turn
bishopMove = isDiagonalMove `AND` tilesBetweenAre emptyTile
                            `AND` allyTile `AND` NOT allyDestination
```

Figure 3.8: Constructing the condition for the moveset of a bishop in chess.

3.2.5 Rules

In the `Game` data type there are three fields containing lists of rules, each serving a different purpose. The fields along with their purpose are as follows:

- `preTurnConditions`: Some games have rules that should be applied before you are allowed to input your turn. An example of this is if the player's turn should be skipped if they have no legal move.
- `rules`: Rules that determine how a turn should be played, what the player is allowed to do, and what happens when a turn is played.
- `endConditions`: Rules that are checked after a turn has been played to see if the game should end. These rules also determine how the game should end, i.e. which player won or if it was a draw.

In order to fully understand rules and how to chain them together to create complex rule sets it will be helpful to look at how the data type is defined. This can be seen in figure 3.9. Notice that almost all constructors use a `Rule` in some way. This means that you can chain different rules together to get more complex behaviour. There are ten constructors in total and their purpose will be explained below.

```
data Rule = Rule         (Update Game)
          | If           (Condition Turn) Rule
          | IfElse       (Condition Turn) Rule Rule
          | Rule `SEQ`    Rule
          | Rule `THEN`   Rule
          | Rule `THEN2`  Rule
          | TurnRule     (Update Turn) Rule
          | IterateUntil Rule (Condition Turn)
          | ForAllDir    [Update Turn] (Update Turn -> Rule)
          | ForEachDir   [Update Turn] (Update Turn -> Rule)
```

Figure 3.9: The implementation of the `Rule` data type.

The first constructor is the most basic one, `Rule (Update Game)`. When applying the rule, the update will simply be applied which will subsequently change the game state.

The second constructor is an if statement: `If (Condition Turn) Rule`. The condition takes a turn as its type parameter and uses that together with `Game` to return a `Bool`. If the condition is met the rule will be applied. However, if the condition is not met the rule will fail, which leaves the game state unchanged. This is important to know for understanding some of the behaviour of the other rule constructors. The following constructor, `IfElse`, is much the same as the previous. The difference is that it has an extra `Rule` that is applied if the condition is not met, which means that an `IfElse` will never fail to apply. An example of using an if statement and combining two conditions can be seen in figure 3.10. If the inputted tile is empty and the tile below it is not empty, a piece is placed. Coincidentally, this is how the rules for "placing a piece" in the game Connect Four are defined [16].

```
rules :: [Rule]
rules = [
    If (emptyTile `AND` tileBelowIsNotEmpty) placePiece
]
```

Figure 3.10: An example of a conditional rule using if statements and combining two conditions.

There are three ways of sequencing rules together, `SEQ`, `THEN`, and `THEN2`. Each of them has a corresponding operator. How the operators differ in behaviour is explained in figure 3.11. In the figure, each operator, `>>>` (`THEN`), `>|>` (`THEN2`), and `>=>` (`SEQ`), is chained between three rules. The second rule fails to apply, which

is when the operators differ in behaviour. The colour of the circles corresponds to different states of the game.

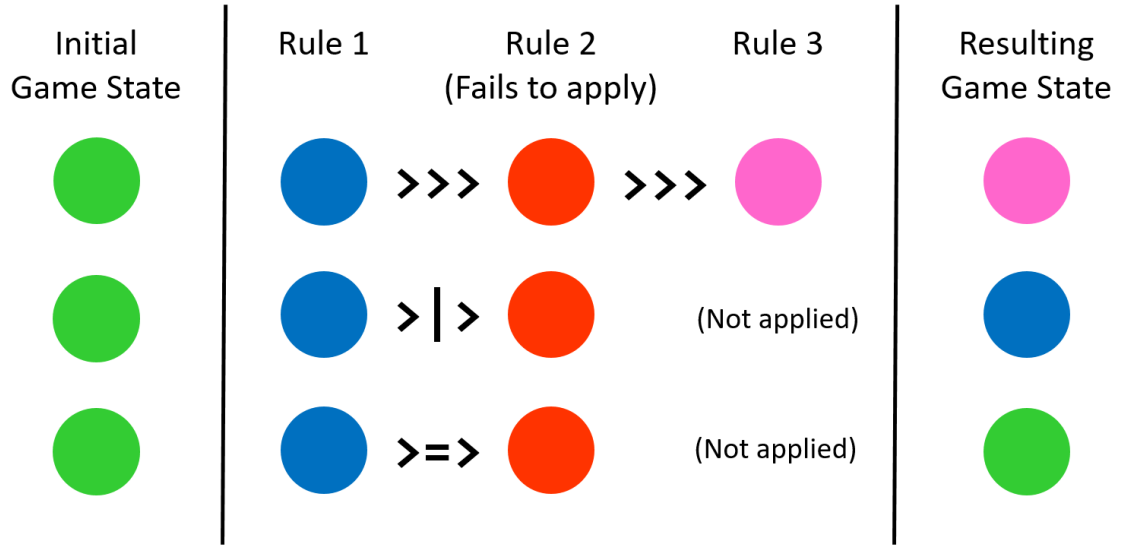

Figure 3.11: Behaviour of the three operators used for sequencing rules when a rule fails.

For `>>>`, every rule in the chain is attempted to be applied. Note that the game state rule three takes in is the blue one, i.e. the last successful state. When a chain of `>|>` includes a rule that could not be applied, it stops and returns the last successful game state. Lastly, chaining multiple `>=>` together means that all of them must succeed for the rules in the chain to be applied. If any of them fail, like in the figure, the whole chain of rules will be discarded.

The next two data constructors for rules are `TurnRule (Update Turn) Rule` and `IterateUntil Rule (Condition Turn)`. The `TurnRule` constructor first modifies the provided turn with a given update, then runs a rule with the newly updated turn. `IterateUntil` applies a `Rule` until a certain `Condition` is met. The specification for `IterateUntil` is as follows:

`IterateUntil r c`:

1. Check the condition `c`.
2. If `c` is true, the rule (`IterateUntil`) succeeds.
3. If `c` is false, try to apply `r`.
4. If `r` fails, the rule fails.
5. If `r` succeeds, go to step 1.

Note that `r` in the example above can be applied multiple times. The constructors `TurnRule` and `IterateUntil` can be combined to apply a rule over an entire row

of tiles. This is the case for a game like Othello, where these are used to correctly update the state of tiles in every direction when a piece is placed.

The last two data constructors are `ForAllDir [Update Turn] (Update Turn -> Rule)` and `ForEachDir [Update Turn] (Update Turn -> Rule)`. These constructors take a list of `Update Turn` and apply each of the updates to a `TurnRule`. This will return a list of rules that each uses different turns. These rules are then chained together with sequencing operators. The `ForAllDir` constructor combines rules with the `>=>` operator and will thus only succeed if all the rules succeed. The `ForEachDir` constructor instead make use of `>>>` and will attempt to apply each of the rules.

3.2.6 Creating a Board Game

In most cases, creating a board game in the DSL will consist of overriding four to six fields in the `game` value. First, a board needs to be initiated. There are two ways this can be done in the library, you can use either the `rectBoard` or `initRectBoard` functions. The former makes every tile empty while the latter enables you to initialise the board with starting pieces.

Then, the pieces and players need to be defined. These should be lists containing one of every player and one of every piece the players can place in the game. Note that, for games with movement only, the pieces can remain an empty list (which is the default) and does not have to be overridden.

The last three fields are the three different kinds of rules described in section 3.2.5. Without defining `rules` you cannot play and without `endConditions` defined, the game can never end. Note that some rules only make sense as an `endCondition`. These are `gameDraw`, `currentPlayerWins`, and `playerWithMostPiecesWins`. These modify the game state and sets `winner` and `gameEnded` accordingly, but should, of course, be combined with other rules containing conditions. The field `preTurnConditions` does not necessarily have to be overridden. See appendix C for a list of all available rules, conditions, and updates included in the library that you can use when implementing games.

3.3 Example Games

During the development, a few games were modelled using the DSL to test its functionality as well as showcase the DSL's capabilities. Three of the implemented games are tic-tac-toe, Othello, and chess. This section will briefly go over the source code for these games.

3.3.1 Tic-tac-toe

How tic-tac-toe has been modelled using the DSL can be seen in figure 3.12. The first step in modelling tic-tac-toe is by overriding the board field in the `game` value. The board is defined as an empty three-by-three grid. Following this, the pieces and players are listed. Lastly, the rules and end conditions are defined.

```
tictactoe :: Game
tictactoe = game {
    board = rectBoard 3 3,
    pieces = [
        Piece "X" (Player "A"),
        Piece "O" (Player "B")
    ],
    players = [
        Player "A",
        Player "B"
    ],
    rules = [
        If emptyTile placePiece
    ],
    endConditions = [
        If (inARow 3) currentPlayerWins,
        If boardIsFull gameDraw
    ]
}
```

Figure 3.12: An implementation of tic-tac-toe using the DSL.

Note the order of the `endConditions`. The way it is specified, if both conditions happened to be true from the same turn only the first end condition will be applied, which, in this case, is very important.

3.3.2 Othello

The Othello implementation can be seen in figure 3.13 and starts similar to that of tic-tac-toe. A difference is how the board is initialised with four pieces in the middle. Other than that, it is the rules that are defined differently.

Here, `preTurnRules` is used to enable skipping turns when a player has no move. To save space and allow more readable code, `othelloRule` has been defined outside the game definition. It is the intricate rule that in all directions, flip enemy pieces to ally pieces if there is an allied piece in a direct line from the placed tile, whilst there are no empty tiles between.

3.3.3 Chess

Chess is a more advanced game compared to the previous two. The implementation is therefore a bit larger and cumbersome, but it should still be easy to understand all parts. It can be found in appendix B. As mentioned previously, this implemented version has some missing rules such as castling and en passant.

The initialisation of the board requires more lines of code because of the 32 pieces having to be listed individually. The biggest difference compared to Othello and

```
othello :: Game
othello = game {
    board = initRectBoard 8 8 [
        ((4,4), Piece "O" (Player "A")),
        ((5,5), Piece "O" (Player "A")),
        ((4,5), Piece "X" (Player "B")),
        ((5,4), Piece "X" (Player "B"))
    ],
    pieces = [
        Piece "O" (Player "A"), Piece "X" (Player "B")
    ],
    players = [
        Player "A", Player "B"
    ],
    preTurnRules = [
        If (NOT playerCanPlace) skipTurn
    ],
    rules = [
        If (emptyTile `AND` changedState othelloRule)
            (placePiece >>> othelloRule)
    ],
    endConditions = [
        If noPlayerHasMoves playerWithMostPiecesWins
    ]
}
othelloRule :: Rule
othelloRule = ForEachDir allDirections
        (replaceUntil enemyTile allyTile)
```

Figure 3.13: An implementation of Othello using the DSL.

tic-tac-toe is the pieces have individual movement rules. This is solved by chaining a lot of rules where the type of piece is checked before testing the validity of the move. For determining the move validity, the `destinationIsRelativeTo` and `tilesBetweenAre` conditions are used extensively. Most of these rules are defined in the code library (see appendix C) with the notable exception of the pawns. The pawns behave differently in certain positions and can only move in one direction which requires a greater set of conditions having to be met.

3.4 Implementation Details

The function that runs a game is the `playGame` function and can be seen in figure 3.14. The function is optional and was created to test the DSL and show it working. It acts as the run function for the type `Game` and sequentially handles the inputs and applies the rules accordingly. The function calls itself until the

game ends by reaching an end condition. Before inputs are received the optional `preTurnRules` are checked. If these rules are successfully applied the `playGame` function is called again with the new game state. To get inputs from the players the functions `getValidInput` and optionally `getValidPiece` are called. These functions are not particularly interesting, they simply ensure that the inputs from the players are in the correct format. The reason why the game has ended is checked twice is to prevent infinite loops that could occur if the rules of the game were incorrectly specified, but this could admittedly have been avoided much more cleanly.

```
playGame :: Game -> IO ()
playGame g = do
  dispFunction g g

  let g' = postPlayTurn nullTurn g
  if gameEnded g' then
    case winner g' of
      Nothing -> putStrLn "Draw!"
      Just p -> putStrLn $ "Player " ++ show p ++ " has won!"
  else do
    let g'' = applyRules nullTurn g preTurnRules
    when (g /= g'') (playGame g'')

    let currPlayer = head $ players g
    putStrLn $ "Player " ++ show currPlayer ++ "'s turn"

    input <- getValidInput g
    piece <- getInputPiece g input

    let newGame = playTurn (Turn piece input) g
    if g == newGame then
      putStrLn "Input move does not follow the rules" >>
      playGame newGame
    else if gameEnded newGame then
      dispFunction g newGame >>
      case winner newGame of
        Nothing -> putStrLn "Draw!"
        Just p -> putStrLn $ "Player " ++ show p ++ " has won!"
    else
      playGame newGame
```

Figure 3.14: The run function for the `Game` data type.

The notable function called in `playGame` is `playTurn` which can be seen in figure 3.15. It is there that the rules of the game are applied. When calling `playTurn` there are two possible options. Either the input is invalid and it simply returns the input game state, signalling to `playGame` that it can fetch another input, or the input is valid and calls `postPlayTurn`. It can be seen in figure 3.15 as well, and takes as

input a game state after the current player has made their turn. It applies the end condition rules and changes the order of the list of players so that the next player will be the first element in the list, thereby changing the current player.

```
playTurn :: Turn -> Game -> Game
playTurn t g | not $ isValidInput t g = g
             | otherwise = postPlayTurn t newGame
    where
        newGame = applyRules t g rules


postPlayTurn :: Turn -> Game -> Game
postPlayTurn t g = newGame {
    players = cyclePlayers $ players newGame
}
    where
        newGame = applyRules t g endConditions
```

Figure 3.15: The `playTurn` and `postPlayTurn` functions.

In order to apply rules, `playTurn`, `postPlayTurn`, and `isValidInput` all call the `applyRules` function that can be seen in figure 3.16.

```
applyRules :: Turn -> Game -> (Game -> [Rule]) -> Game
applyRules t g f =
    foldl (\g' r -> fromMaybe g' $ runRule r t g') g (f g)
```

Figure 3.16: Function that applies a list of `Rule` to the game.

This function iterates over a list of type `Rule` and applies each of those to the game through `runRule`. The result of applying the first rule is sent in as an argument to applying the next rule, and so on. If applying a rule ever yields `Nothing`, which happens when the rule fails to apply, the previous game state is used as an argument to the next rule instead.

3.4.1 Run Functions

As explained in section 2.2.2, a run function is a function that interprets the parse tree generated by inputting values into the corresponding data types and computes the result. There are three run functions tied to the corresponding data types `Update`, `Rule`, and `Condition`. The function type signatures are listed below, along with a short description.

`runUpdate :: Update t -> Turn -> t -> t`
Run function for the update type.

```
runRule :: Rule -> Turn -> Game -> Maybe Game
```
If the rule is applied, meaning that the game state has changed, it returns `Just game`, otherwise `Nothing` is returned.

```
runCondition :: Condition Turn -> Turn -> Game -> Bool
```
Evaluates a condition to true or false.

3.5 Testing

Unit tests were created during the process to verify that parts of the library worked as intended. The Haskell packages QuickCheck and hspec were used to aid the test creation process. QuickCheck is a well-established framework for automatically testing functions on random inputs to verify that given properties hold [17], while hspec is a DSL used for defining tests [18], making them easier to manage.

QuickCheck was not used as much as was first planned, simply because a lot of the functions turned out to be difficult to test with random inputs. Testing games with randomised sets of rules could be a good way to get rid of errors tied to the structure of the code but this was not in the scope of the project. Even if you had a fixed game with fixed rules, randomising other parameters like turns would still be difficult. How would you determine the effect of an arbitrary turn so that you could verify that the output was correct?

Instead, the majority of testing opted towards functionality in complete games with determined turns, without using QuickCheck. This was done for the games tic-tac-toe, Othello, and chess. Tests for these games were constructed by devising a sequence of turns that should put the game state in particular ways. For instance, when the board is full in tic-tac-toe with neither player having three in a row, the game should end in a draw. A test function would play turns that should end in a draw and check that this was the case. Of course, this only verifies that a particular chosen sequence of turns ends in a draw. However, it still gives at least some assurance that the draw functionality for tic-tac-toe works as intended. Doing tests like this does not verify individual functionality on their own, but rather whole games with many individual parts working together. This, together with the fact that these tests were very easy to put together, meant that it was these kinds of tests that gave the most value during development.

4

Discussion

This chapter will discuss various parts of the project and DSL. Some evaluation if the DSL was a success and reflections on that, together with reflections on important choices made and their consequences. Then, a section discussing how the DSL can improve in the future. Lastly, the impact that a language for board games can have on society.

4.1 Changes from the Planning Phase

During the planning phase we wanted the DSL to support the game Ludo. In fact, Ludo was the second game we had as a goal for the DSL to support instead of Othello as is described in section 1.2. The reason for the change was mainly based on time. Quite late in the development phase, we realised that the current rule system, which was at that point neither shallow nor deep embedded, was simply not good enough. The DSL could be used to create the specific games we added support for, but provided little to no ability to create customised games, making it a terrible DSL. For instance, an implemented function `changeSurrLines` was used for the whole logic of flipping pieces correctly for the game Othello to work. But, as said in the introduction, for it to be a DSL that could support a large number of board games, there would have to be a way of creating complex rules (such as `changeSurrLines`) using simple components and features in the DSL. To achieve this, we therefore decided to change the rule system completely to a deep embedded version using new types for `Rule`, `Condition`, and `Update`, with corresponding run functions.

The changes to the rule system turned out to be a good one, as the resulting DSL was both easier to use and provided support for a much larger set of games. Othello's `changeSurrLines` could now be implemented using abstract constructs instead of the game-specific function. After this change, we could draw three conclusions:

1. A lot of time had passed, meaning that we would probably not have time to implement chess if we had begun working towards games like Ludo.

2. There would still have had to be major changes to the DSL to allow for die rolls, supporting multiple pieces on the same tile, and allowing for games to provide a given path pieces should follow.

3. Othello is a game with complex rules, well worthy of being a sub-goal to aid in the fulfilment of the purpose.

For these reasons, Ludo was changed to Othello as a sub-goal for the DSL development process. Instead of the major structural changes that iteration two would include, they were kept in mind for future expansions, which can be read in section 4.4.

4.2 Evaluation

The purpose of the project was fulfilled, at least to a degree. The DSL can be used to model both simple placement games such as tic-tac-toe as well as more complex ones like Othello. It is, however, questionable how easy it is for a user to create games on their own. This will be discussed more extensively in section 4.2.1. As mentioned previously, not all chess rules could be implemented and the reasons why this is the case will be discussed in section 4.3.2. But in short, this is because the library is missing necessary functionality that would need major changes to the DSL to get working.

Still, there are playable versions of the games tic-tac-toe, Othello, and chess, along with a few other board games. It should be possible to use the DSL for creating a large number of other board games, as well. Unfortunately, time ran out before we could properly evaluate if this is the case.[1] The games supported at the moment are those that can be constructed within our current limitations described later in this chapter under section 4.3.2. This mainly includes board games with a rectangular board and actions tied to the current turn. But if memory restrictions are ignored, the DSL should be able to be used to model an infinite number of (although perhaps not that varied) board games.

As is shown in table 4.1 the amount of code that needs to be written decreases significantly with the use of the DSL. The main caveat to the table that should be noted is that the functions that are used by the rules are not accounted for. The implementation using the DSL in the figure accounts for the assignment of values for the game's starting format, and any rules that are merely combinations of rules provided in the `Lib.hs` file included in the library. If the rule functions necessary for creating a game are already provided, the creation of the game becomes a relatively quick process as long as the user understands the way the DSL works. And even when the functions need to be created, the user merely needs to make a function that creates a certain state change. They do not need to create any types or logic for running the game, provided they do not want to create a separate game-running function aside from the included `playGame` function.

[1] This could have been evaluated by randomly selecting board games within our scope and seeing if it is possible to implement without editing the DSL as well as how easy it is. Ideally, this would be done as a case study to decrease the chances of getting a biased evaluation.

Game	DSL Implementation	Haskell Implementation
Tic-tac-toe	19	77
Mnk-game	21	108
Connect Four	19	130
Othello	30	152
Chess	64	N/A

Table 4.1: Table showing lines of code per game implementation, with and without using the DSL.

4.2.1 Usability

The goals for the usability of the language that were set were that it should be expressive, and the easier it is to make games, the better. Expressiveness is at least partially a subjective measure. However, what can be said about the DSL is that it should not be too inexpressive. It uses if statements similar to that of other languages, but without the need for braces. You can create almost sentence-like rules as can be seen in figures 3.10, 3.12, and 3.13. Unfortunately, not every rule is equally expressive. The rules for creating the movement of pawns in chess, which can be seen in appendix B, are quite verbose. This is because the DSL does not allow the rule to be modelled in a more generic way than it currently is. A fix for this can simply be trying to abstract the movement of pieces further. It has to also be said that the level of expressiveness is tied to that of the host language, Haskell. Because of Haskell, we are forced to create if statements in certain ways (without braces), and there are limits to how nesting and chaining rules can be written. Now, you have to sometimes use the `$` operator or surround rules with brackets to get the correct order of execution.

Another thing that had to be taken into consideration during the whole course of development was how easy the language is to use. Implementing games with simple rules and few pieces such as tic-tac-toe or Connect Four requires minimal work from the user and not much thought. For these games we are content with the way games are specified. This can be extended to Othello as well. The Othello rules are significantly more complicated than those of tic-tac-toe but can be specified in only a few lines of code. When it comes to chess there are certain parts that can be considered successful and certain parts that could be considered as failures. The rules of chess can be written in a surprisingly small amount of code, especially considering how many rules are tied to pawns alone. The problem is that when looking at the rules it is quite unclear what they do.

The choice of not having a map or similar structure that maps a set of rules to each piece might be confusing to the user of the language. As it is currently a condition is used to make a rule specific to a certain piece. This solution probably results in less code that needs to be written, but also a less intuitive system. The balance between the amount of code, and the intuitiveness and readability of the code is important and our language generally trends towards the former.

Defining the board is tedious and takes up many lines of code, most of them only being copied and slightly changed. Fortunately, solving this problem would not require too much work. There is a function that was used for simplifying testing called `parseBoard`, that, with some small changes, could be useful for initiating complicated boards like chessboards. It takes a nested list of pieces and creates a board with the pieces in the same location as they were in the lists. Changing this function to take in a nested list of strings instead would make it require fewer lines of code but it would also require the information about which string matches which piece. To solve this the `Game` data type could be changed to take a function with the type signature `a -> [Piece] -> Board`.

One other thing that had to be considered was how many conditions that should be generalised. On one hand, generalising a condition to encompass more is most certainly useful. On the other hand, the generalisation will often lead to the condition being less comprehensible and requiring more code.

4.3 Reflections

As mentioned, the functionality provided according to the planning phase has had to be revised due to structural problems in the coding phase. The problem was mainly the lack of planning before starting the development. Because we selected a few games and implemented them in Haskell first, before trying to develop a DSL for them, instead of creating a DSL we more or less created a library that could only be used for those selected games. We had little to no thought about abstracting and generalising rules for a long time. When we finally realised what needed to change to enable the functionality we wanted, a lot of time had passed. This has had the consequence that less functionality than initially planned has been provided. But, by introducing more deep embedding into our structure we were able to produce a DSL with a better level of abstraction when it comes to the rules. The syntax also became more logical which included similarities to if statements many developers using the DSL should be familiar with. The first thing we would do differently is to study more specifically what a DSL "should" look like. That way, we could perhaps have gotten started with a deep embedded rule system quicker and thereby have more time to refine it, as the current one has flaws.

4.3.1 Type of Embedding

In this project, we chose to implement a deep embedded DSL. The choice was primarily made with the end-user in mind. A deep embedded DSL enables the users to add their own interfaces, possibly even writing their own interpretation of different rules and pieces. If we had instead used shallow embedding we would limit the user with our interpretations, giving the user less freedom of the types of game they can create. This is because of the nature of shallow embedding, that they only allow one interpretation.

Towards the end of the project, we realised that the number of constructors of the

type `Rule` was a limiting factor for what games it can model. Adding more would have been easier to do with a shallow embedding. The drawback would be the fact that only one interpretation could be used which we did not want to enforce.

The results show the difficulty in choosing the type of embedding. If we choose to use shallow embedding we are left with data types that are not easily modified. The changes in the data types here have a large effect on the implementation, and changing them leads to changing all the functions using a specific data type too.

If we look at the deep embedding chosen for this project we got a more modular solution but had to spend more time on creating a structure of the functions suitable for deep embedding. This shows us that this approach leads to a more modular DSL. It is easy to add new functionality without having to change existing code. The functions themselves can be written to represent one action, and then chained together to construct more complex moves.

4.3.2 Limitations

The games supported by our DSL are games based on the following restrictions:

Board The board must be a two-dimensional rectangular board where all tiles are either empty or occupied by a single game piece.

Movement The movement must be based on calculations using the grid of the board.

Turn A turn must be one of two actions: either the placement of a single piece or the movement of a single piece. Turns can be skipped if conditions are met, meaning that a player can make multiple turns in a row. However, the support for this is limited. Retrieving input on how the turn will play out can happen once during each player's turn before the main rules are applied.

Rules The rules must be based on chained deterministic actions tied to the current turn and game state. Rules based on previous states of the game and nondeterministic rules are not supported in the DSL.

The restrictions above are based on keeping the DSL simple and having a working version at the end of the project. If there was time for reducing the limitations mentioned above, we would have had to create vast changes to how the DSL is structured or how it works internally.

As stated before, the complete set of chess rules could not be implemented. En passant and castling are moves that require previous moves to determine if they are allowed, which is not possible without saving turn history or the ability to save states in pieces. Also, because of the restrictions on a turn, castling could not be implemented as that would involve moving more than one piece. Lastly, the whole rule system does not currently have the right level of abstraction to be able to model such advanced concepts as stalemate or determining if a move puts your king

in check and thereby not allow it. To do this, you would have to have more control over what can happen during your opponent's turn and allow or disallow certain moves based on that.

4.4 Future Expansions

The DSL is far from perfect. This section will discuss how the DSL can improve, both by simple additions and bigger, structural changes.

4.4.1 GUI

Terminals limit the way pieces can be represented, and it can be hard to differentiate between different pieces only from the one character that represents them. This limits the user experience of games that can be created. Fortunately, it is possible to connect a GUI with the DSL. By creating your own `playGame` function you can use a GUI framework, like Threepenny-GUI [19], to connect the board game to a proper user interface. This would allow for a better user experience along with all benefits of using a DSL for creating board games.

4.4.2 DSL Improvements

The simplest and perhaps most important improvement to the library would be to expand the collection of conditions, rules, and updates available. These are directly tied to the set of supported games and creating more would lead to more games being supported. If a condition needed for a particular board game is missing from the library there are three options for the user — they move on and do not use our library, they exclude the condition or use one that is similar enough, or they write their own condition in Haskell code. All options are undesirable, and to avoid these problems as much as possible there need to be as many conditions, rules (and ways of creating rules), and updates as possible. Another approach would have been to provide a more generalised method for building rules and conditions, maybe by using the same idea we use for combining them, but this could lead to a more complex DSL which might require large structural changes. A combination of the two parts would probably be the best solution; providing a set of predefined rules and conditions and providing the option to construct compound rules using the method above. If time had allowed, we could have reduced this problem by looking at more board games within our scope and developing the DSL further if they could not be modelled with the DSL.

In the DSL itself, there are some limitations. An `Update` is currently defined to always take a `Turn` as an argument, but this should not always have to be the case. There are, in fact, a few functions in the provided library that creates an `Update` without making use of the `Turn` argument at all. It would instead make more sense to change the type of the `Update` constructor to a more generalised version, something similar to `Update (t -> t)`. This would open up more options for what the constructor is capable of, like for example making an `Update` function

that takes the union between two boards. The same can be said about the way we define a `Condition`. Currently, a `Condition` always takes a `Game` as an argument, but several functions do not make use of this forced argument. A better solution would be to remove the `Game` argument and replace it with a polymorphic type, a possible solution would then look like `Condition (a -> Bool)`. A fair few changes would need to be done to support this change, as all the library functions and most of the run functions are dependent on the current types of `Update` and `Condition`.

Another problem with a simple solution is that only rectangular boards can be created. Hexagonal boards, boards with holes in them, and other boards other than rectangular cannot currently be created. To solve this the `Tile` data type could be updated according to figure 4.1. Not many changes would be needed to accommodate this, but additional pattern matching would be needed and `runCondition` would need to be changed to always return false if a selected tile is equal to `Void`.

```
data Tile = PieceTile Piece Pos | Empty Pos | Void Pos
```

Figure 4.1: A potential improvement to the tile data type

Currently, only one piece can be on a tile at a time. An obvious way to change `Tile` so that any number of pieces can be on the same tile is for each tile to store a list of pieces that are on it. This, however, would make it more difficult to distinguish between games that support multiple pieces on the same tile and those that do not within the DSL, as the behaviour of turns would differ between them. One potential solution to this could be to parameterise `Game` with what kind of tiles it should consist of — single piece tiles or tiles allowing multiple pieces.

The way input and output, IO, is handled at the moment is not ideal. Currently, all inputs are handled in the `playGame` function (figure 3.14) and the output is handled by a display function defined inside the `Game` data type (figure 3.1). Extracting all IO handling from `playGame` would remove the need for the IO monad which would simplify the code. This would also make it easier to create new functions for handling IO operations. By removing the display function (`dispFunction`) from `Game` we simplify the type and make it more focused on the core structure. The display function is not part of the DSL semantics and should therefore not be in the `Game` data type. By moving the display function it can be combined with the input handling function, making it a function which handles all IO. This would also make connecting a GUI to the games easier.

If one were to look for larger and more structural changes that would expand the scope of the games that could be constructed using the DSL there are many options. Due to how games are implemented and run in our library, many games simply cannot be modelled. One missing feature that reduces the number of possible games is die rolls. Randomness is a key concept that is present in many games. Among these games are classic board games such as *Settlers of Catan* [20] and *Monopoly* [21]. As it stands right now these classics cannot be made and a clear improvement to the language would be to support die rolls. To support die rolls there would need

to be a way for the user to specify in the rules that a die should be rolled as well as how the result of the roll should be used. Perhaps the `Action` data type could be expanded to encompass more actions other than `Place` and `Move`, such as `DieRoll`, `DrawCard`, and `PassTurn`. If drawing cards or adding items were to be added, each player would also need to have an inventory where all their items are stored. It should also be noted that Settlers of Catan is a game of imperfect information which would still lay outside the current scope of the DSL if the changes above were to be implemented. If one were to want to support games of imperfect information, the DSL would ideally have to be played on a server or using peer-to-peer so that different players can play using their own devices and see the information exclusive to them, without anyone else seeing it.

One thing that was noticed when chess was made was that only taking inputs from the user once per turn limited the kind of rules that could be created. In chess, when a pawn reaches the opposite end of the board the player must replace said pawn with either a knight, rook, bishop or queen. This means that the player has to make an additional choice after moving a piece. This cannot be done with our current DSL so the rule had to be simplified to the pawn always being promoted to a queen. To allow for the full rule to be modelled, some larger structural changes would have to be done. Instead of taking the input from the player each turn at a fixed point in time, there would have to be a way for the user to specify in the rules when an input should be taken and what kind of input it expects.

These structural changes could also solve another issue with how games are run. As it functions in the current version of the code, the logic for the `preTurnRules` is inside `playGame`. Ideally, all logic would be removed from the `playGame` IO-function (figure 3.14) to be able to create tests for all aspects of a board game as well pushing users to write their own user interfaces. If a new and more all-encompassing approach to rules were to be taken, then the rules for Othello could look like what is shown in figure 4.2. This approach essentially has all `preTurnConditions` before the rest of the rules and the function `getPlaceInput` decides when and what kind of input is needed for the rest of the rules. Doing it this way would simplify both the `playGame` function, extracting the input handling to elsewhere, and the `Game` data type, as `preTurnConditions` could be removed completely. This would be an overall improvement to the DSL. However, as Haskell is a pure functional language, getting input inside the rules would probably mean using IO monads in the rules data type, which could make the code less manageable.

4.4.3 Uses in Research

In the introduction, we claimed that one could write a program that computes the possibilities of each player winning from a given state of a board game. To do this, all possible permutations of each player's turns would need to be computed as well as the outcome of the game for each permutation. To get all possible permutations it is necessary to get all possible turns for a player from a certain game state. The logic for obtaining all locations that a player can place a piece on already exists as it is necessary for the rules of Othello. Extending this to return a list of all possible

```
If (NOT playerCanPlace) skipTurn >>> getPlaceInput >>>
    If (emptyTile `AND` changedState othelloRule)
                (placePiece >>> othelloRule)

...

othelloRule = ForEachDir allDirections
        (replaceUntil enemyTile allyTile)
```

Figure 4.2: An example of how the rules for Othello could be better specified.

turns instead would be simple. The logic for obtaining all possible turns involving a move would be more complex. There would have to be a function that loops through all occupied tiles on a board and for each of these tiles, it would have to loop through all tiles on the board to see if the piece from the first tile can be moved there. Doing this would be computationally expensive, especially for games with larger boards and more possible moves. It would be possible nonetheless and most certainly an interesting use of the DSL.

4.5 Societal Considerations

In the creation of a tool for creating board games, there are not many ethical pitfalls to look out for. In other words, there are not many concerns about how creating this DSL would have an overall negative effect on the world. There are, however, conversations to be had about what utilities digitising board games could bring to areas like analysis of game theory and accessibility.

4.5.1 Analysing Games

A good example of how bringing board games into a digital space can give way to a lot of interesting theory is the history of chess AI. Ever since 1997, the best chess player in the world has been an AI. These AIs have forwarded the chess scene immensely, allowing the game at a top-level to be understood much more deeply and played more optimally than before these innovations were made [22]. Having more board games be digitised gives way for more of these games to be more understood, and lets us understand how to solve these logic puzzles much quicker through the use of artificial intelligence.

Creating a DSL for board games might be the first step in making an AI able to play a large number of different board games, not just specialising in one or a few. In turn, this could create a sort of race where people must learn the new optimal strategies to have a chance at competing with AIs, and in consequence, other players. This would drive up the competition for the best board game players in the world while also enabling finding the most optimal strategies through playing and analysing the games.

4.5.2 Accessibility

There are several concerns about accessibility in board games today, both in terms of accommodating physical disabilities like visual and cognitive impairment, and in terms of being inclusive towards underrepresented groups of people [23]. Some of these accessibility issues and how digitising board games can benefit them will be discussed in this section. For certain concerns about visual impairment digitising the board games could prove quite useful. An example of this is that several games symbolise several mechanics, such as player adherence, through colour. This can be an issue for people with colour blindness. However, with a digital version of the board game, changing colours to be inclusive of certain versions of colourblindness can be a quick and easy fix with a basic understanding of the game's code, as opposed to a physical game where you might be tasked with painting game objects to adhere to this inclusion.

For people with more encompassing visual impairment, such as partial or total blindness or severe issues with differentiating visual information, there are other issues to be addressed. Certain problems regarding this can be addressed to some extent through digital versions of these board games. For instance, one way people with total blindness play board games is to say what moves they want to make and have someone else move the pieces for them. This can also be the case for people with certain disabilities regarding motor functions. In a digital setting with a clear input structure, this is made easier as there is little to no wiggle room for interpretation of what they mean when expressing the inputs for a turn. Another issue blind people face with board games is that there is a lot of emphasis in several board games to know the board state in every instance. Since they cannot see the board state this has to either be memorised throughout the whole game, or communicated aloud any time something happens in the game (usually some combination of the two). With a digital version of these board games a user interface that reads out the board state quickly to the user, akin to blind accessibility programs for reading text on phones and computers, could be implemented. This type of interface, and many others, are possible to implement because of the deep embedding. Thanks to the ability to have multiple interpretations of the same game, the game can be adapted to meet all needs.

People with memory impairments are generally accommodated rather well by board games, as most games can be played from the perspective of the current board state with no regard for what has happened earlier in the game. There are, however, games with mechanics that relate to memorising previous moves. Making board games digital can be helpful in these cases, as it is possible to keep old moves available to look back upon in the game interface.

Another issue board games face is a lack of representation. This issue is not always present, as in many cases characters and objects are very abstracted. In many cases in board games, a player is represented by a text character or a figurine of an object or something of the like. However, where human characterisations do appear there is often an overrepresentation of cis white male characters, which could make the hobby seem aimed at that group, rather than be inclusive to all. The representation

issue would not be solved directly by moving board games to a digital platform. But akin to how the issue colour blind people face with certain games, creating playable character pieces would be easier for a digital game than a physical one. Creating a digital character piece is more intricate than changing the colours of certain game elements, but the workload is still lessened for diversifying these elements in a digital setting.

5

Conclusion

Board games are entering a new era — the digital era. For thousands of years, games have been an integral part of human life. This project wanted to simplify the creation of board games in a digital setting with the help of an embedded domain-specific language. The eDSL's goal was the ability to model a large number of board games while being easy to use.

This thesis detailed the development process of the eDSL and the theory behind it. Also included is a guide on how to create board games using the eDSL as well as a list of all tools available to the user of the library. A few games showcasing the language's abilities and its inner workings are presented alongside the guide. The DSL is evaluated as a success, being able to model many board games. Although, how easy it is to use is a bit unclear. The thesis concludes with a discussion about choices made, potential improvements and expansions, and the possibilities digitising board games using a DSL can have. The complexity of what we needed to model decided the type of embedding since there are no optimal choices to be made as shown in our project. Some of the potential improvements include updates to the `Game`, `Update`, and `Condition` data types. Finally, digitising board games can enable great improvements to the accessibility of board games.

Now it is time for you to continue the work towards a more digital gaming experience.

Bibliography

[1] Meilan Solly. "The Best Board Games of the Ancient World". In: (2020). URL: https://www.smithsonianmag.com/science-nature/best-board-games-ancient-world-180974094/.

[2] *Ludii*. URL: https://ludii.games/index.php (visited on 22/04/2022).

[3] Simon Marlow et al. "Haskell 2010 language report". In: (2010). URL: http://www.haskell.org/.

[4] Tomaž Kosar Mernik et al. "A preliminary study on various implementation approaches of domain-specific language". In: *Information and Software Technology* 50.5 (2008), pp. 390–405. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2007.04.002. URL: https://www.sciencedirect.com/science/article/pii/S0950584907000419.

[5] Andy Gill. "Domain-specific Languages and Code Synthesis Using Haskell: Looking at embedded DSLs". In: *Queue* 12.4 (2014), pp. 30–43. ISSN: 1542-7730. DOI: 10.1145/2611429.2617811. URL: https://doi.org/10.1145/2611429.2617811.

[6] Jeremy Gibbons. "Functional Programming for Domain-Specific Languages". In: *Central European Functional Programming School, CEFP 2013*. Ed. by V Zsok, Z Horvath and L Csato. Vol. 8606. Lecture Notes in Computer Science. Babes-Bolyai Univ, Cluj-Napoca, Romania; Eotvos Lorand Univ, Budapest, Hungary. 2015, pp. 1–28. ISBN: 978-3-319-15940-9; 978-3-319-15939-3. DOI: 10.1007/978-3-319-15940-9_1.

[7] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison-Wesley, 2010. ISBN: 9780321712943.

[8] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2006.

[9] Marjan Mernik, Jan Heering and Anthony M. Sloane. "When and how to develop domain-specific languages". In: 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[10] Eelco Visser. "WebDSL: A Case Study in Domain-Specific Language Engineering". In: *Advanced Data Mining and Applications*. Advanced Data Mining and Applications, 2008, pp. 291–373. ISBN: 0302-9743. DOI: 10.1007/978-3-540-88643-3_7.

[11] Pierluigi Riti. "External DSL". In: *Practical Scala DSLs: Real-World Applications Using Domain Specific Languages*. Apress, 2018, pp. 59–69. ISBN: 978-1-4842-3036-7. DOI: 10.1007/978-1-4842-3036-7_4. URL: https://doi.org/10.1007/978-1-4842-3036-7_4.

[12] Josef Svenningsson and Emil Axelsson. "Combining Deep and Shallow Embedding for EDSL". In: *Advanced Data Mining and Applications*. Advanced Data Mining and Applications, 2013, pp. 21–36. DOI: 10.1007/978-3-642-40447-4_2.

[13] Eb'rahim Durosimi. *Game Theory*. Electronic Book. 2013. URL: https://ujuzi.pressbooks.com/.

[14] William Poundstone. *Prisoner's dilemma: John von Neumann, game theory, and the puzzle of the bomb*. Doubleday, 1992.

[15] Oleksandr Nevskiy et al. *Mysterium*. Board Game. 2015.

[16] Howard Wexler and Ned Strongin. *Connect Four*. Board Game. 1974.

[17] Koen Claessen and John Hughes. "QuickCheck". In: *ACM SIGPLAN Notices* 35.9 (2000), pp. 268–279. ISSN: 0362-1340. DOI: 10.1145/357766.351266.

[18] Simon Hengel. *Hspec: A Testing Framework for Haskell*. 2022. URL: https://hspec.github.io/ (visited on 22/04/2022).

[19] Heinrich Apfelmus. *threepenny-gui: GUI framework that uses the web browser as a display*. 2021. URL: https://hackage.haskell.org/package/threepenny-gui (visited on 11/05/2022).

[20] Klaus Teuber. *Settlers of Catan*. Board Game. 2015.

[21] Lizzie Magie and Charles Darrow. *Monopoly*. Board Game. 1935.

[22] Patrick Gebhardt. *The history of chess AI*. Electronic Article. 2019. URL: https://blog.paessler.com/the-history-of-chess-ai (visited on 03/03/2022).

[23] Michael James Heron et al. "Eighteen Months of Meeple Like Us: An Exploration into the State of Board Game Accessibility". In: *The Computer Games Journal* 7.2 (2018), pp. 75–95. ISSN: 2052-773X. DOI: 10.1007/s40869-018-0056-9.

[24] Alejandro Russo. *Implementation: deep embedding*. Requires login. 2022. URL: https://chalmers.instructure.com/courses/17400/pages/implementation-deep-embedding?module_item_id=241032 (visited on 25/04/2022).

[25] Alejandro Russo. *Implementation: shallow embedding-2*. Requires login. 2022. URL: https://chalmers.instructure.com/courses/17400/pages/implementation-shallow-embedding-2?module_item_id=241031 (visited on 25/04/2022).

A

Example of a DSL using deep and shallow embedding

This is an eDSL made in Haskell by Alejandro Russo for the course Advanced Functional Programming (TDA342) at Chalmers. The data is a signal and the main purpose is sampling it. It is a good example showcasing the differences between a deep (figure A.1) and a shallow (figure A.2) embedding. In the deep embedding, all signal data is stored in a data type and is interpreted by the `sample` function. The shallow embedding interprets the data while creating the data type. This makes the sampling function very simple compared with the one using the deep embedding.

```
type Time = Double
data Signal a where
  ConstS :: a -> Signal a
  TimeS  :: Signal Time
  MapT   :: (Time -> Time) -> Signal a -> Signal a
  (:$$)  :: Signal (a -> b) -> Signal a -> Signal b

constS = ConstS
timeS  = TimeS

sample (ConstS x)  = const x
sample TimeS       = id
sample (f :$$ s)   = \t -> sample f t $ sample s t
sample (MapT f s)  = sample s . f
```

Figure A.1: An example of a deep embedded DSL. Used with permission. [24]

```haskell
type Time = Double
newtype Signal a = Sig {unSig :: Time -> a}

constS :: a -> Signal a
constS x = Sig (const x)

timeS :: Signal Time
timeS = Sig id

($$) :: Signal (a -> b) -> Signal a -> Signal b
fs $$ xs = Sig (\t -> unSig fs t  (unSig xs t))

mapT :: (Time -> Time) -> Signal a -> Signal a
mapT f xs = Sig (unSig xs . f)

sample :: Signal a -> Time -> a
sample = unSig
```

Figure A.2: An example of a shallow embedded DSL. Used with permission. [25]

B

Chess modelled with the DSL

This appendix contains the code of the game chess, figure B.1, modelled with the DSL. To make everything fit, `chessBoard` and `chessRules` are shown in their own figures, figure B.2 and figure B.3 respectively.

```
chess :: Game
chess = game {
  board = chessBoard,
  players = [
    Player "White",
    Player "Black"
  ],
  rules = chessRules,
  endConditions = [
    If (pieceNotOnBoard "k" `OR` pieceNotOnBoard "K")
        currentPlayerWins
  ]
}
```

Figure B.1: An implementation of chess using the DSL. Code for `chessBoard` can be found in figure B.2 and `chessRules` can be found in figure B.3.

```
chessBoard = initRectBoard 8 8 [
    ((1,7), Piece "p" (Player "White")),
    ((2,7), Piece "p" (Player "White")),
    ((3,7), Piece "p" (Player "White")),
    ((4,7), Piece "p" (Player "White")),
    ((5,7), Piece "p" (Player "White")),
    ((6,7), Piece "p" (Player "White")),
    ((7,7), Piece "p" (Player "White")),
    ((8,7), Piece "p" (Player "White")),
    ((1,8), Piece "r" (Player "White")),
    ((2,8), Piece "h" (Player "White")),
    ((3,8), Piece "b" (Player "White")),
    ((4,8), Piece "q" (Player "White")),
    ((5,8), Piece "k" (Player "White")),
    ((6,8), Piece "b" (Player "White")),
    ((7,8), Piece "h" (Player "White")),
    ((8,8), Piece "r" (Player "White")),

    ((1,2), Piece "P" (Player "Black")),
    ((2,2), Piece "P" (Player "Black")),
    ((3,2), Piece "P" (Player "Black")),
    ((4,2), Piece "P" (Player "Black")),
    ((5,2), Piece "P" (Player "Black")),
    ((6,2), Piece "P" (Player "Black")),
    ((7,2), Piece "P" (Player "Black")),
    ((8,2), Piece "P" (Player "Black")),
    ((1,1), Piece "R" (Player "Black")),
    ((2,1), Piece "H" (Player "Black")),
    ((3,1), Piece "B" (Player "Black")),
    ((4,1), Piece "Q" (Player "Black")),
    ((5,1), Piece "K" (Player "Black")),
    ((6,1), Piece "B" (Player "Black")),
    ((7,1), Piece "H" (Player "Black")),
    ((8,1), Piece "R" (Player "Black"))
]
```

Figure B.2: How the chessboard for the game chess is defined using the DSL.

```
chessRules = [
    If (allyTile `AND` NOT allyDestination) $
        IfElse (pieceEqualToEither ["H", "h"] `AND` isKnightMove) movePiece $
        IfElse (pieceEqualToEither ["k", "K"] `AND` isKingMove) movePiece $
        IfElse (pieceEqualToEither ["q", "Q"] `AND` isQueenMove) movePiece $
        IfElse (pieceEqualToEither ["b", "B"] `AND` isBishopMove) movePiece $
        IfElse (pieceEqualToEither ["r", "R"] `AND` isRookMove) movePiece $
        IfElse (pieceEqualTo "p"              `AND` isWhitePawnMove)
            (movePiece >|> If (pieceDestinationBelongsToRow 1) (convertToPiece "q")) $
        If     (pieceEqualTo "P"              `AND` isBlackPawnMove)
            (movePiece >|> If (pieceDestinationBelongsToRow 8) (convertToPiece "Q"))
]

isWhitePawnMove = (destinationIsRelativeTo (0,-1) `AND` emptyDestination)
     `OR` ((destinationIsRelativeTo (1,-1) `OR` destinationIsRelativeTo (-1,-1)) `AND` enemyDestination)
     `OR` (pieceOriginBelongsToRow 7 `AND` destinationIsRelativeTo (0,-2)
           `AND` emptyDestination `AND` tilesBetweenAre emptyTile)

isBlackPawnMove = (destinationIsRelativeTo (0,1) `AND` emptyDestination)
     `OR` ((destinationIsRelativeTo (1, 1) `OR` destinationIsRelativeTo (-1, 1)) `AND` enemyDestination)
     `OR` (pieceOriginBelongsToRow 2 `AND` destinationIsRelativeTo (0,2)
           `AND` emptyDestination `AND` tilesBetweenAre emptyTile)
```

Figure B.3: How the chess rules are defined using the DSL.

C

Library functions

Below is a list of every rule, condition, and update included in the library, together with ways of initialising the board and the display function.

C.1 Boards

```
rectBoard :: Int -> Int -> Board
```
Creates an empty rectangular board.

```
initRectBoard :: Int -> Int -> [((Int, Int), Piece)] -> Board
```
Creates a rectangular board with pieces in certain locations.

C.2 Rules

```
placePiece :: Rule
```
Places a piece in a certain position on the board.

```
movePiece :: Rule
```
Move a piece to an absolute position on the board.

```
gameDraw :: Rule
```
A `Rule` for a draw.

```
currentPlayerWins :: Rule
```
A `Rule` for the turn player winning.

```
playerWithMostPiecesWins :: Rule
```
A `Rule` for when the player with the most pieces out on the board wins.

```
skipTurn :: Rule
```
Passes the turn according to the order described in the game state.

```
convertToPiece :: String -> Rule
```
A rule that places a specified piece at a tile specified by the turn input.

C.2.1 Rule Utility

```
doUntil :: Rule -> Condition Turn -> Update Turn -> Rule
```
Iterate a `Rule` until a `Condition` is met over an `Update Turn`.

Example usage:

```
doUntil (If emptyTile placePiece) enemyTile
```

The example replaces the next tile, if empty, with an ally tile. If the condition fails before reaching an enemy tile, the result is ignored.

```
replaceUntil :: Condition Turn -> Condition Turn -> Update Turn ->
                Rule
```
Replace a tile until another tile in the specified direction.

Example usage:

```
replaceUntil enemyTile allyTile
```

The example replaces every enemy tile until the first ally tile in the specified direction. If another tile is reached before the end condition is met, the result is ignored.

C.3 Conditions

```
trueCond :: Condition Turn
```
A condition that is always `True`.

```
falseCond :: Condition Turn
```
A condition that is always `False`.

```
pieceIsAtPos :: Pos -> Condition Turn
```
A condition which is true if a given piece is at a given location.

```
pieceOriginBelongsToRow :: Int -> Condition Turn
```
A condition which is true if a piece at the first position of a move turn exists in a given row.

```
pieceDestinationBelongsToRow :: Int -> Condition Turn
```
A condition which is true if a piece at the first position of a move turn exists in a given row.

```
boardIsFull :: Condition a
```
Checks if the board is full.

```
changedState :: Rule -> Condition Turn
```
A condition for checking if a `Rule` would change the state of the board.

```
emptyTile :: Condition Turn
```

A `Condition` for checking if the current tile is empty.

```
allyTile :: Condition Turn
```
A `Condition` for checking if the piece (on given tile) belongs to the current player.

```
enemyTile :: Condition Turn
```
A `Condition` for checking if the piece (on given tile) doesn't belong to the current player.

```
pieceEqualTo :: String -> Condition Turn
```
A `Condition` for checking if a predescribed piece is equal to a given piece on the board.

```
pieceEqualToEither :: [String] -> Condition Turn
```
A `Condition` for checking if the piece on the tile has any of the identifiers in the given list.

```
emptyDestination :: Condition Turn
```
A `Condition` for checking if the destination tile is empty.

```
allyDestination :: Condition Turn
```
A `Condition` for checking if the destination belongs to the player.

```
enemyDestination :: Condition Turn
```
A `Condition` for checking if the destination belongs to the enemy.

```
destinationIsRelativeTo :: (Int, Int) -> Condition Turn
```
Checks if the destination is x steps left/right and y steps up/down compared to original position.

```
isDiagonalMove :: Condition Turn
```
A `Condition` for checking if a given move follows a diagonal path.

```
isStraightMove :: Condition Turn
```
A `Condition` for checking if a given move is a straight line.

```
pieceOnBoard :: String -> Condition Turn
```
A `Condition` for checking if the piece is on the board.

```
pieceNotOnBoard :: String -> Condition Turn
```
A `Condition` for checking if the piece is not on the board. Inverse of `pieceOnBoard`.

```
noPlayerHasMoves :: Condition Turn
```
Returns `True` if no player has any valid moves, `False` otherwise.

```
playerCanPlace :: Condition Turn
```
A `Condition` for checking if the current player can place a piece anywhere on the board.

```
inARow :: Int -> Condition Turn
```
Checks if the board contains a given number of pieces in a row in any orientation

(vertical, horizontal, diagonal).

`tileBelowIsNotEmpty :: Condition Turn`
A condition for checking if the tile below another tile is empty.

`tilesBetweenAre :: Condition Turn -> Condition Turn`
A condition for checking if another condition applies to all tiles between the origin and destination pos.

The following conditions are for different moves in chess. Note that the conditions are not exclusive to the game chess, but are just descriptive names for the conditions.

`isKnightMove :: Condition Turn`
A condition for determining if the turn is a knight move in the game chess.

`isKingMove :: Condition Turn`
A condition for determining if the turn is a king move in the game chess.

`isRookMove :: Condition Turn`
A condition for determining if the turn is a rook move in the game chess.

`isBishopMove :: Condition Turn`
A condition for determining if the turn is a bishop move in the game chess.

`isQueenMove :: Condition Turn`
A condition for determining if the turn is a queen move in the game chess.

C.4 Updates

`allDirections :: [Update Turn]`
A list containing all directions.

`straightDirections :: [Update Turn]`
A list containing all straight directions.

`diagonalDirections :: [Update Turn]`
A list containing all diagonal directions.

The following updates are for updating a turn in specific directions, mainly for use with `IterateUntil`. The name of the functions signify the direction.

`turnUp :: Update Turn`

`turnLeft :: Update Turn`

`turnRight :: Update Turn`

`turnDown :: Update Turn`

```
turnUpLeft :: Update Turn

turnUpRight :: Update Turn

turnDownLeft :: Update Turn

turnDownRight :: Update Turn
```

C.5 Display functions

```
prettyPrint :: Game -> IO ()
```
Prints a board in the terminal. It's pretty.