Aalborg Universiy

Computer Science

# EPIC TITLE

Authors

Sebastian Wahl
Simon Buus Jensen
Elias Khazen Obeid
Niels Sonnich Poulsen
Kent Munthe Caspersen
Martin Bjeldbak Madsen

Supervisor

Nicolaj Søndberg-Jeppesen

February 2013 - May 2013

DEPARTMENT OF COMPUTER SCIENCE
AALBORG UNIVERSITY

AALBORG UNIVERSITY
STUDENT REPORT

**Title:**

**Abstract:**

**Theme of project:**

Design, Definition and Implementation of Programming Languages

**Project period:**

P4
2013 February 4 – 2013 May 29

**Project group:**

d402f13 (d402f13@cs.aau.dk)

**Participants:**

Sebastian Wahl
Simon Buus Jensen
Elias Khazen Obeid
Niels Sonnich Poulsen
Kent Munthe Caspersen
Martin Bjeldbak Madsen

**Supervisor:**

Nicolaj Søndberg-Jeppesen

**Ended:**

2013 May 29

# Preface

# Contents

CHAPTER 1

# INTRODUCTION

For this project we've been working with developing a programming language that can be used to describe board games. The specific topic is to design, define, and implement a programing language for generic game playing. This means that we must analyse board games and try to outline which aspects of board games our language must be able to describe. These are aspects like how many players can play? What is the board size? Turns? Possible actions? Winning condition?

## PROJECT GOALS

So what is the goal of this project? Why do we need a new programming language for generic game playing - is it not possible to code games in C or Java? Yes it is possible to code games in already existing programming languages but the goal of the project is that the students gain knowledge of important underlying concepts in the world of programming languages. How are these concepts derived? How are they formally described and represented in an implementation?

Obviously, all software is written in some kind of a programming language and compiled or interpreted so it can be executed. Design, definition and implementation of programming languages is a central topic of Computer Science. By gaining a better understanding of these topics the student will be able to grasp the possibilities of different programming languages and programming paradigms and what their differences are.[8, p. 22] We will discuss different paradigms in section 2.8.

The goal is that:

> the student must learn how to design and implement a programming language and how this process can be supported by formal definitions of the languages syntax and semantics and the techniques and methods to construct a translater for the language.[8, p. 22]

This report presents and documents the process and work of which we've been through to reach this goal.

## WHY BOARD GAMES?

So what do board games have to do with designing, defining, and implementing a programming language? If you're able to describe how board games work on a very generic and general level, it would theoretically be possible to abstract away from that to describe all games in general.

If a language purely concentrates on allowing the programmer (i.e. game designer) to express how his game works, it would be quicker and easier to pick up and develop shorter and more precise programs rather than having to reimplement everything from scratch in an already existing high-level language. Furthermore, data structures and special statements specifically designed to help define board games would greatly increase the readability and writability of such a program.

A language designed with board games in mind would also allow the game designer to, relatively quickly, explore new ideas for a board game with a simple implementation. He/she could then efficiently modify the code according to a new rule or idea, and the implementation would stay exactly the same. If the language also took multiple platforms into account, it would open up the possibility to run the same game across multiple devices.

This could enable AI creation for the language that understands the rules, so you can test an early implementation of your game without the need of other human players.

An example could be four-person chess. If you already had an implementation of chess in the programming language set up with a board and separate rules for each piece, then it could be as simple as changing the piece location and player count (and maybe editing the rules for one or two of the pieces so it's a little more fair) followed by running the program again.

Programming languages exist to create programs that express algorithms to control the behaviour of machines. Most board games, as we will demonstrate, have specific rules and exact winning conditions to follow, which can be described to a very detailed degree.

Different categories of board games exist and board games can be placed into different genres, such as: strategy, alignment, chess variants, paper-and-pencil, territory, race, trivia, wargames, word games, and dozens of others. Obviously some games can overlap genres. Chess is an example of this. It is obviously a chess variant and is very strategy-heavy.

Before we begin to develop our programming language we have to do some research about board games, the different programming paradigms, and then we will give an overview of what a compilers job really is, then we will present our findings about scanning and parsing methods, followed by a comparison of compilers and interpreters, and finally we will define what a game simulator is and why we need one. When the analysis is done we present our problem statement at the end of this chapter.

# Chapter 2

# Analysis

## 2.1 Board game analysis

One may wonder what a board game really is. Could it just be any game containing some kind of a board? If so, would Trivial Pursuit be a board game and what about the game Twister, where you have to place your hands and/or feet on a spot marked with a particular color on a sheet - or board, as you could call it. Most people have a mental model of a board game that does not include games like Twister. Here is one definition of a board game [10]:

> "A board game is a game played across a board by two or more players. The board may have markings and designated spaces, and the board game may have tokens, stones, dice, cards, or other pieces that are used in specific ways throughout the game."

The definition above is very broad and will to some extent allow a game like Twister to be categorized as a board game. All kinds of things like cards and dice can be part of a board game, but one board game designer may also be able to invent a new and yet unseen widget, which he wants to include in his board game. A programming language that makes it possible to describe any board will cover a very broad category of games. You could argue that it would actually cover all games that can be made, since even a first person shooter could technically be played across a board. With such a broad definition, a programming language that aims to make the programming of board game easier, will be a programming language that aims to make almost everything easier. If a programming language aimed to make the programming of only a very specific kind of board games easier, there might be many things that can be optimized compared to existing general purpose programming languages.

Due to there being very different definitions of board games depending on the source, we decide to make our own definition of board games that our programming language should be able to describe. This is partly to help further define and outline a specific genre or type of board games that we find interesting and relevant. We also decide to do this to keep the language as simple and straightforward as possible, while still keeping the things we choose not to include in the back of our minds, to make including these features possible in a later stage of development.

To create our own definition of board games, we look at three existing and popular, yet relatively different board games and sum up what features they have in common.

## 2.2 Chess, Kalah and Naughts & Crosses

In the following sections a detailed analysis of the three games: Chess, Kalah, and Naughts & Crosses will be performed. The reason why we pick these three games is because they are among those we have personal interest in,

Figure 2.1: *The board game chess with the pieces in start position.*

and feel are essential for a generic board game programming language's possible descriptions of games. Therefore we want to dig deeper into the details of the components of these games (e.g. the pieces, the board, the squares, etc.) to gain a better understanding of which features are needed in GARRY. The respective history and basic rules of the games and other related information will not be included in the analysis, since this has no relevance for gaining understanding of how our programming language should be designed.

### CHESS

Chess is a board game of two oppenent players. It's a turn-based game which means one player makes a move, then the other player makes a move, then the first player makes a move and so on. Chess is played on a board of $8 \times 8$ squares. The squares are typically black and white, but can be any two colors (see figure 2.1). The squares can only contain one piece at a time, unlike games like Mancala and Backgammon. Each player has a total of 16 pieces: 8 pawns, 2 knights, 2 bishops, 2 rooks, a queen and a king. Each type of piece has unique ways to move. For instance a pawn can move only one square vertically forward or one square diagonal when capturing an enemy piece. A rook can move unlimited squares either forward or backward (vertical movement), or to the right or to the left (horisontal movement). This separates it's pieces from a lot of other board games where all pieces have the same abilities e.g. Naughts and Crosses, Mancala, Ludo, Backgammon etc.

Cut to the bone the game goes as follow: When a game starts the pieces are in their respective starting positions as seen in figure 2.1. The player with the white pieces always makes the first move, and after that the players shifts in turn in which clever moves are beeing taken and pieces are beeing captured until one player has checkmated the other - and the game is over. The checkmate situation is obtained when the king piece is in a position to be captured and cannot escape from capture. [2].

Special situation and moves. In chess there are numerus special situations and moves which doesn't follow the normal pattern of chess. Earlier we mentioned that a pawn can only move only one square vertically forward or one square diagonal when capturing an enemy piece. But this is not always true. If the pawn is in it's respective starting position it can move either one **or** two squares vertically forward. After moving from it's starting position it can only move one square forward the rest of the game. Another special move is the move called "castling". This move allows a player to move two pieces in one turn (the king and one of the rooks). But to do the move several conditions needs to be met. First: the move has to be the very first move of the king and the rook, second: there can't be any pieces standing between the king and the rook and third: there can't be any opposing pieces that could capture the king in his original square, the squares he moves through or the square he end up in [2].

Possible requirements

- pieces with different movement abilities.
- A square board with a number of squares in it.
- A winning condition - when the king has been checkmated.
- A starting state - how the pieces are placed on the board before the game's very first move.

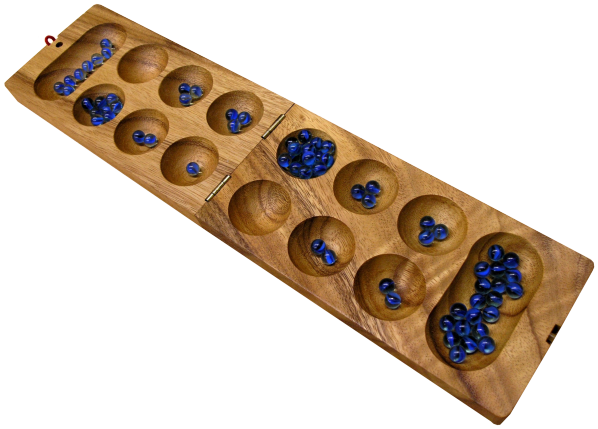

Figure 2.2: *The board game Kalah*

- Defining special situations like the "castling"

### KALAH

> "The object of the game is to capture more seeds than one's opponent. At the beginning of the game, three seeds are placed in each house. Each player controls the six houses and their seeds on his/her side of the board. His/her score is the number of seeds in the store to his/her right. Players take turns sowing their seeds. On a turn, the player removes all seeds from one of the houses under his/her control. Moving counter-clockwise, the player drops one seed in each house in turn, including the player's own store but not his/her opponent's. If the last sown seed lands in the player's store, the player gets an additional move. There is no limit on the number of moves a player can make in his/her turn. If the last sown seed lands in an empty house owned by the player, and the opposite house contains seeds, both the last seed and the opposite seeds are captured and placed into the player's store. When one player no longer has any seeds in any of his/her houses, the game ends. The other player moves all remaining seeds to his/her store, and the player with the most seeds in his/her store wins. It is possible for the game to end in a draw, with 18 seeds each."

When comparing Kalah and the game elements involved to other board games, some interesting requirements for a generic board game programming language able to describe this particulary game arises. The requirements described here consider seeds as pieces and the houses as squares, which makes it easier to compare the requirements of Kalah to the requirements of other board games later.

- Squares can contain an arbitrary number of pieces
- Making a move can be considered as simply choosing a square
- A turn may contain more than one move
  - If the last piece of your move is put on a specific square, you can make another move.
- A square can be related to a player.
  - A winner is found based on who has most pieces in a special square.
  - On your turn you can only choose a square beloning to you.
- Squares can be related to other squares.
  - You place pieces on squares counter-clowise.
  - If you place a last piece on an empty square, the nearest square controlled by the opponent is emptied.

- The game ends when some condition is met
  - All pieces are on one specific square.
  - If you place a last piece on an empty square, the nearest square controlled by the opponent is emptied.
- When the game ends, a winner is found
  - The player with most pieces wins
- Only one type of pieces
- The number of pieces on a square determines how long a move you can make

### Naughts & Crosses

Naughts & Crosses is like Kalaha and Chess a game of two opposing players. The game is played on a board with $3 \times 3$ squares. Each square has the same properties with no acceptions or special situations to ruin this.

possible requirements

- Turn-handling.
- A start condition

### Similarities and differences

From looking at the three board games (Chess, Kalah, and Naughts & Crosses) many different game elements have been recognised. For a programming language that allows all of the three games to be described, all of the game elements in each of those must be able to be designed in the programming language. Some of the elements are very similar, in which case a more generalised description has been provided. For example, the rook piece and the bishop piece in chess do not have equal moves but their moves have been generalised to just *movement by patterns*. The general game elements in the earlier mentioned board games can be seen here:

- The game has a single board.
- The board contains squares in a two-dimensional grid.
- The game contains one or more types of pieces.
- The game has an initial setup.
- There are no dice.
- There are no cards.
- A piece can be either on the board or off the board.
  - If a piece is on the board it is positioned on one specific square.
  - A piece that is off the board may be put on the board.
- A piece belongs to a player.
- A pattern may consider the position of occupied squares, empty squares or other pieces on the board in relation to a specific square or a specific piece on the board.
- A piece has a set of moves which might be an empty set. The possible moves may be based on a pattern.
- A piece may or shall be removed / exchanged based on a pattern.
- At any given time, just one player has the turn.
- If a piece is owned by a player, only that player can use its moves.
- A player's turn can consist of more than one move.
- A player can win if the game is in one or more specific states, which can consider a pattern.
- A game can be a tie if the game is in one or more specific states, which can consider a pattern.

The above-mentioned list will be used to set up the requirements for the design of our programming language. Herein lie a few design restraints that prevent an implementation of quite a few different kinds of games. Granted, many games have these things, but we are deciding to ignore these to make our grasp of implementing a programming language more realistic, while still allowing the programmer to implement a large collection of board games. Monopoly or any form of card games are an example of restraints we have added to our definition of a board game.

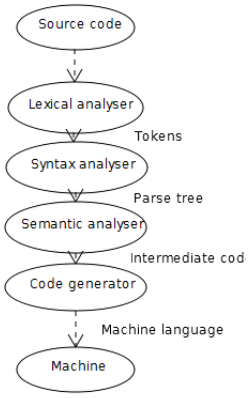


Figure 2.3: *The different phases of a compiler. Based on Sebesta et al.[7] p. 46, Figure 1.3*

This has allowed us to create our own definition of board games that our language design will reflect. Our definition is:

> "Board games are games played on a single rectangle-shaped board with any amount of players taking turns in placing or moving either one or multiple pieces. Pieces have movement patterns to describe the way they behave on the board. There are rules that define the patterns and exchanging of pieces on a board. Winning or tying is achieved by placing the game in a certain state. Board games do not have dice or cards involved."

This definition attempts to be relatively specific, but still leaves room for interpretation, considering the fact that there are thousands of different types of board games with many different elements mixed in.

## 2.3 Compiler overview

A programming language can be implemented by writing a compiler that translates programs into machine language, which a computer can execute directly [7, p. 44]. If the program to be translated is not valid, the compiler should give an error message that best possible helps the programmer identify what error he has made. A good compiler will try to fix the error and continue compilation, so even further errors can be identified, providing the programmer a complete list of errors that can handled all at once instead of one per compilation. Translating one language to another is typically not a simple task, therefore it is often split into different phases, which is shown by figure 2.3 and will be clarified in the following.

### Lexical analysis

The lowest level syntactic units of a language is called lexemes. A language's formal description does not often include these. They are instead described by a lexical specification, regular expressions i.e., separated from the syntactic specification[7, p. 135]. Typical lexemes for a programming language includes integer literals, operators and special keywords like *if* and *while*. If both *$a* and *$b* are lexemes describing a variable and *102* and *42* are lexemes describing an integer, then *$a* and *$b* or *102* and *42* can typically be used interchangeably and still give a meaningful program. Therefore the lexemes are grouped into tokens. The name of a variable or the value of an integer is preserved when tokenising. The tokens are an abstraction that makes it easier to analyse if correct syntax of the language. An example of the grouping of lexemes into tokens can be seen by table 2.1. After the lexical analysis an input stream of characters has been converted to an output stream of tokens.

| Lexemes | $a | = | 3 | $b | + | 4 | $a |
|---|---|---|---|---|---|---|---|
| Tokens | var | assign | int(3) | var(b) | plus | int(4) | var(a) |

Table 2.1: *Lexemes and their corresponding token group.*

## Syntax analysis

All languages whether natural or artificial is a set of strings of characters over some alphabet. There are rules for how the strings can look that are in a language and how they can be combined. The lexemes described how the strings can look and now the tokens are useful when analysis how the lexemes can be combined. The rules can be specified formally to describe the syntax of a language[7, p. 135]. A common way to describe a language's syntax is by a formal language-generation mechanism (also called grammars or context free grammars). By describing a grammar that can generate all possible strings in a language, the language has also been formally described. Backus-Naur Form is such a mechanism which in the 1950's became the most widely used method for describing programming language syntax[7, p. 137]. The BNF contains a set of terminals and a set of non-terminals. The terminals are the tokens from the lexical analysis. The non-terminals all have a set of productions, from which a mix of terminals and non-terminals can be derived from. A start production specifies a single non-terminal, from where all syntactically valid strings that are in the language can be derived from by using the production rules until only a sequence of terminals (the tokens) are left. The syntax analysis takes a sequence of tokens as input and tries to create a set of derivations from the start symbol that creates the given sequence of tokens. If success, the input has been parsed and the parse tree is kept for later analysis. The parse tree is the information concerning how the start symbol was derived into the sequence of tokens, which yields a tree structure. This tree is called an abstract syntax tree.

$$
\begin{array}{lcl}
\textit{program} & \rightarrow & \texttt{"print"}\textit{expr} \\
\textit{expr} & \rightarrow & \texttt{"("}\ \textit{term}\ \texttt{")"}\ \textit{operator}\ \texttt{"("}\ \textit{term}\ \texttt{")"} \\
\textit{operator} & \rightarrow & \texttt{"="} \\
 & | & \texttt{">"} \\
 & | & \texttt{"<"} \\
\textit{term} & \rightarrow & \textit{number} \\
 & | & \textit{expr} \\
\textit{number} & \rightarrow & \textbf{any number}
\end{array}
$$

## Semantic analysis

Not all characteristics of programming languages are easy to describe with a BNF and some even cannot be described using a BNF. If a programming language allows a floating-point value to be assigned to an integer variable but not the opposite, this *can* be expressed with a BNF but if all such rules should be specified in the BNF, it would increase the size of it remarkably. With increased size, the formal description gets more clumsy to look at and also increases the risk that an error is contained in the BNF. The rule that all variables must be declared before being used is impossible to express in a BNF. That would require the BNF to remember things, particularly those variables it had seen before, which it cannot. The problem of remembering things also shows up when we start to concern about scope rules. Typically, a variable declared in one scope cannot be used outside that scope. The BNF cannot describe such problems that we describe as static semantics rules. It is named static because the analysis required to check the specifications can be done at compile-time rather than runtime[7, p. 153]. In this semantic analysis phase, the compiler can check for type rules by starting to decorate the parse tree from the syntactic analysis with types. If the non-terminal *expr* derives the terminal sequence *int plus int semicolon*, it can be decorated with the *int*-type, and the analysis can proceed further up the tree and check that the type of the *expr(int)* is legal. If the *expr(int)* is derived from a *expr* $\rightarrow$ *expr(int)* + *expr(bool)* production, the static semantic rules must determine if the programs semantic is wrong or it the boolean value can be converted to the integer values zero or one.

## Code generation

Every compiler must focus the translation on the capability of a particular machine architecture. The targeted architecture can be virtual such as the Java Virtual Machine. Generally speaking, the code generation phase translates the

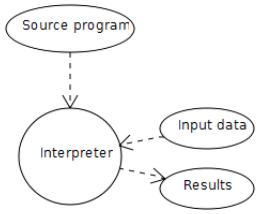


Figure 2.4: *The different phases of an interpreter. Based on Sebesta et al.[7] p. 48, Figure 1.4*

program into instructions that are carried out by a physical processor. Whether the architecture is virtual or real the program code must be mapped into the processors memory. Typically, the overall translation is broken into smaller pieces, where smaller subtrees of the abstract syntax tree are translated into executable form one at a time. However, there many things that must be considered when translating, i.e. **instruction selection**, which concerns how an intermediate code representation from the abstract syntax tree is to be implemented. There are many different ways to implement the same functionality, but some might be carried out faster than other. The code generation phase must also deal with problems concerning **register allocation** and **code scheduling**. Register allocation is concerned with effectively using the registers so moving the same variables between registers and memory is minimised[3, p. 521]. The code scheduling is an important aspect with pipelined processors. The aim is to produce instructions that executes in a way such that the pipelined execution will not have to stall unnecessary[3, p. 551]. Some of the problems associated with the pipelined execution is solved by move apart instructions that will interlock[3, p. 552].

### Interpretation

A pure interpretation of a program lies at the opposite end (from compilation) regarding to methods of implementation. With this approach, which can be see, on figure 2.4, no translation is performed at all. An interpreter is interpreting a program written in the targeted language. It acts like a virtual machine which instructions are statements of high level language. By purely using interpretation, a source code debugger can easily be implemented. Various errors that might occur can once they are detected easily refer to which place in the source code that caused the error. The debugging is eased because the interpreter works like a software implementation of a virtual machine, thus the state of the machine and the value of a specific variable can be outputted at any time when requested. This will of course lead to the disadvantage that an interpreter uses more space than a compiler. Further more, the execution speed of an interpreter is usually 10 to 100 times slower than that of a compiler [7, p. 48].

The compiling or interpreting approach can be combined to form a hybrid implementation system. This method is illustrated in figure 2.5, where a program is compiled into an intermediate code which is then interpreted. By using this approach, errors in a program can be detected before interpretation which can save much time for a programmer. A great portability can also be achieved when using hybrid system. The initial implementation of Java was hybrid and allowed Java to be compiled to an intermediate code that could run on any platform which had an implementation of Java Virtual Machine[7, p. 50].

## 2.4 Context-Free Grammars

Context-free grammars (CFGs) are used to specify the syntactical structure of programming languages. Throughout the report we will be using context-free grammars to represent our programming language. In this section we define what a context-free grammar is.

A context-free grammar is a collection of substitution rules (or productions) constituted by nonterminals and terminals that describe the construction of a language. Any language that can be generated by a CFG is called a context-free langauge. A CFG is a 4-tuple $(V, \Sigma, R, S)$:[5, p. 100]

- $V$ is a finite set called the nonterminals
  - To create a string, we think of the nonterminals as variables

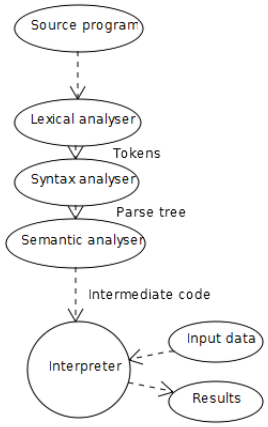


Figure 2.5: *The different phases of a hybrid implementation systems. Based on Sebesta et al.[7] p. 49, Figure 1.5*

- $\Sigma$ is a finite set, disjoint from V, called terminals
  - The terminals cannot be the same as the nonterminals
  - This set can be thought of as the alphabet
- $R$ is a finite set of productions
  - Each production consist of a nonterminal and a string of nonterminals and terminals
  - The nonterminal and the string is seperated by an arrow or a |
- $S \in V$ is the start symbol (a nonterminal)
  - This is the first nonterminal at the left-most top of the grammar

In the following section we give an example of a CFG.

### Production

It is easier to understand what a CFG really is by showing an example. The following is an example of a CFG, which we call *acb*:

$$\begin{aligned} A &\rightarrow aAb \\ &\mid B \\ B &\rightarrow c \end{aligned}$$

In the example we have two nonterminals; A and B, and three terminals; a, b, and c. The production for nonterminal A states that A can derive the string "aAb" or the string "B", and the nonterminal B can derive the string "c". When a nonterminal is present in a string they are substituted with their own production. For instance the string *aaacbbb* can be derived from the CFG we called *acb*. It is not really possible to create a lot of different strings with *acb*. It is only possible to create strings with an equal amount of a's and b's with a c in between them.

### Derivation

The sequence of substitutions needed to obtain a string from the CFG is called a derivation.[5, p. 100] A derivation of the above given string *aaacbbb* is:

$$A \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaaBbbb \Rightarrow aaacbbb$$

The derivation starts with the start symbol (which is the left hand side of the production) which is substituted with its substitution rule (which is the right hand side of the production). The nonterminals are substituted until there are no more left. When the string only contains terminals the derivation is complete.

Now that we have an understanding for CFGs we can go on to the next topic. The following sections will cover the topic about parsers; what they are and how they are generated.

## 2.5 Compiler and interpreter

Along with the design of the programming language Garry, we also want to make it possible for programmers who write games in Garry to actually play them afterwards. There are a number of ways we can make that happen. It can be translated to platform dependant machine instruction using a compiler, or it can be parsed and executed on-the-fly using an interpreter.

### Compilation

With compilation, an executable would be created for a specific platform which contains all the code required to play the game. Since games have common aspects, a game engine containing all the common aspects such as user interface, AI and/or network play would most likely be written. This engine would then be included directly in the executable.

An obvious disadvantage is that the executable is platform dependant and it would therefore be necessary to develop a new compiler for each platform we want to support. On the other hand knowing the specific platform makes it possible to create optimized code which runs faster.

Instead of compiling to native machine code, it could be compiled to an intermediate format such as Java bytecode which is supported on many platforms. While Java bytecode is interpreted and therefore slower, modern interpreters uses sophisticated methods such as Just-In-Time compilation (JIT) which recompiles it into native machine code. This process of course adds an overhead, however the speed differences are not that great anymore[4].

### Interpretation

An interpreter takes the original source code directly to parse and execute it in one step. This separates the game code from the forehand mentioned engine and requires the end user to get both the interpreter and the actual game. Different games written in our language would then use the same copy of the interpreter, instead of having a copy of the engine for each executable. This separation will be further explored in section 2.5.

The execution speed will however suffer and while techniques such as JIT exists to improve this, it is beyond the scope of this project. **TODO:** can we justify this?

The execution speed is not critical however. Processors nowadays are fast and executing tasks such as calculating moves would most likely finish so fast that you wouldn't be able to notice the difference between a compiled and a interpreted version. One task where speed does become important is artificial intelligence (AI), which would be used to create an virtual opponent controlled by the computer. The virtual opponent becomes smarter the more turns it can look ahead, however the computation complexity is high and a doubling of speed is quite noticeable when the time to make a move is reduced from 2 minutes to one.

An inherent consequence of interpretation is that the original source code is available for everyone which obtains the game. This can discourage developers which intends to sell their game, as it is easy for everyone obtaining a copy to make derivatives of it. Others will find it as an advantage as they can fix errors, add new gameplay elements or use it as a base for a completely new game.

### Separation of game and engine

Keeping the game and the engine separated opens up for the possibility of changing the game engine while still being able to use the same game.

One major advantage is that it is possible to update the engine and in result, update all your games. An update which improves the graphics or add new features such as network support would work with older games instantly, without having to wait for the developer to update it. If the developer no longer maintains the game, a updated version might never come out.

The disadvantage with this is however that the responsibility for maintaining compatibility is moved from the developer of the game to the developers of the engine. A game developer can simply change his program so it works with a new engine, however the engine developers would have to support games written for every version released.

#### Compiled plug-in

It is possible to achieve this using compilation too. The game could be compiled to a plug-in, which the engine loads dynamically.

### Security

When compiling to a native instruction set, we have access to every instruction on that platform, also potentially unsafe instructions. Even though our language does not include features which makes use of those instructions, it is possible to use code injection on the compiled code to make it execute any code. This way you could create a trojan horse, which appears to be a normal game but might do malicious actions in the background. For example, it could randomly delete files from the users document folder each time he won.

With interpretation we define ourselves which instructions exists and therefore can chose only to include instructions which we know are safe. Even if we decide to allow certain questionable actions, since it is not executed directly on the CPU and instead goes through our interpreter, we can provide a sand-boxed environment which restricts the actions to only allow a safe subset. For example, we might provide access to the file system, but only allow file deletion in the games own directory.

### Intermediate format

A middle step between compiling and interpretation is to compile to a intermediate format which is then interpreted. The intermediate language could be more low-level which would make it possible to optimize the code for higher efficiency.

The intermediate format could be stored as an archive file which contains not only the code, but also sounds, images and other resources required to play the game. This would allow for easy distribution of a game in Garry. The source code would not be available like with a compiled game, however a package format could allow to optionally include the original source if the developer wants to share.

Using an intermediate format however means that you need to create a compiler, an interpreter and the intermediate language, which in turn is a significant larger amount of work.

### Summery

A compiler can make faster code, however an interpreter allows us to fine-tune security considerations. Separating the game engine and game code gives a significant improvement for both methods. Creating an intermediate language can give some advantages over a purely interpreted language, however it also requires more work to develop.
«««< HEAD

## 2.6 Parser

A parser is the component or algorithm that controls whether or not the source code of a given application is set up syntactically correct according to the programming language it is written in. This process is called the syntax analysis. In short, the parser takes as input the stream of tokens produced by the scanner, and checks if the given sequence of tokens corresponds to the program language's grammar. If this is the case the sequence is syntactically correct else a syntax error has occurred, which has to be dealt with before the compiler or interpreter can proceed. In this section we are going to make an analysis of some different types of parsers. There exists two main approaches to

parsing, namely top-down parsing and bottom-up parsing, which have very different ways of dealing with the parsing process. A variety of parsers derive from each approach. For instance the LL parser, the LR parser and the LALR parser, which we are going to analyse. Further more we are going to look at different methods for making parsers, more specifically we are going to analyse some of the pros and cons against writing the parser by hand versus using parser generator tools to generate it.

### LL-PARSERS AND LR-PARSERS

The LL-parsers is a family of parsers which derives from the top-down parsing approach. In this approach the parsers starts at the start symbol of a grammar and through a series of leftmost derivations tries to match the input string, if possible. On the opposite the LR-parsers starts with the input string and through a series of reductions tries to get back to the start symbol.

In both approaches a parse tree is build. The root of the parse tree contains the start symbol, and the branches the input string. The LL-parsers build the parse tree from the root down to the branches - hench the name top-down parsing. While LR-parser builds the parse tree from the branches up to the root - hench buttom-up. A simple illustration of a parse tree can be seen in figure ref(fig:XXXX). Here the parser gets the input string "int + int". And it's based on the simple context free grammar seen in table reftable:XXXX).

The LL-parsers has two actions: predict and match. The predict action is used when the parser is trying to guess the next production to apply in order to get closer to the input string. While the match action eats the next unconsumed input symbol if it corresponds to the leftmost predicted terminal. These two actions are continuosly called until the entire input string has been eaten and thereby has been matched.

The LR-parsers also has two actions: the shift action and the reduce action. The shift action adds the next input symbol into a buffer for consideration. The reduce action reduces a collection of nonterminals and terminals into a nonterminal by reversing a production. These two actions are continously called until the input string is reduced to the start symbol. citehttp://stackoverflow.com/questions/5975741/what-is-the-difference-between-ll-and-lr-parsing. An example of a LR(2)-parser and LL(1)-parser in action can be seen in table 2.6 and table 2.6.

| | Production | Input | Action |
|---|---|---|---|
| 1 | S | int + int | Predict S → E |
| 2 | E | int + int | Predict E → T + E |
| 3 | T + E | int + int | Predict T → int |
| 4 | int + E | int + int | Match int |
| 5 | + E | + int | Match + |
| 6 | E | int | Predict E → T |
| 7 | T | int | Predict T → int |
| 8 | int | int | Match int |
| | | | Accept |

| | Workspace | Input | Action |
|---|---|---|---|
| 1 | | int + int | Shift |
| 2 | int | + int | Reduce T → int |
| 3 | T | + int | Shift |
| 4 | T + | int | Shift |
| 5 | T + int | | Reduce T → int |
| 6 | T + T | | Reduce E → T |
| 7 | T + E | | Reduce E → T + E |
| 8 | E | | Reduce S → E |
| | S | | Accept |

Compared to the LR-parser, the LL-parsers are much easier to write by hand and understand. They are not as powerfull as the LR-parsers in the sense that they do not accept as many grammars, and the LR-parser are generally parsing faster. ****** KILDER og flere sammenligninger følger *******

## 2.7 Requirements

This section presents a set of requirements for this project. The purpose of a requirements specification is to make sure that the final product does what is was intended to do and meets the specified requirements. It is used throughout the development phases and requirements are added as the project moves along and new challenges arise.

The requirements specification consists of three main points: functional requirements, non-functional requirements and solution goals. Functional requirements define what the final system should be able to do. Non-functional requirements define different constraints and boundaries for the entire project. Lastly, solution goals are overall requirements that help us define the correct solution to our problem statement [1].

**Functional requirements:**

- The programming language will be used to program board games **(Must have)**
- The programming language should as a minimum make it possible to implement chess and the special rules of chess **(Must have)**
- It must be possible to play the created board games in a graphical simulation **(Must have)**
- The programming language must be designed in such a way that it will be possible to keep track of the move history **(Must have)**
- It must be possible to play over a network in the simulation **(Wont have)**
- Players must have the possibility to undo a move in the simulation **(Could have)**

The list of requirements also have **non-functional requirements** which is split into two topics - performance limitations and project limitations.

**Performance limitations:**

- The programming language must be easy to learn and use by novice programmers **(Must have)**
- Programmers must be able to be able to implement board games with relatively few lines of code **(Must have)**
- The programming language must not be an extension of another programming language **(Must have)**
- The board games should as a minimum consist of two players **(Must have)**
- The source code of a single board game must be written in one file **(Must have)**
- The formal definition of the programming language must be described in Extended Backus-Naur Form (EBNF) **(Must have)**
- The programming language must either be compiled (by a compiler) or interpreted (by an interpreter) **(Must have)**

**Project limitations:**

- The programming language must be functional and operable no later than 29th of May 2013
- There group has approximately 20 hours per week to work on the project
- The project is limited by the group members' skills in the design development of programming languages
- The project (and hence programming language) must have a catchy name and logo

**Solution goals:**

- The programming language must make it easy and quick for programmers to develop board games which are within the scope of the defined problem statement **(Must have)**
- The board games must be playable on different operating systems **(Should have)**

**TODO:** Sum up here. What are the most important points?

=======

## 2.8 Paradigms

A programming paradigm describes a method and style of computer programming. Some of the primary paradigms are imperative, object-oriented, functional and declarative programming. While some programming languages strictly follow one paradigm, there are many so-called multi-paradigm languages, that implement several paradigms and therefore allow multiple styles of programming. Examples of multi-paradigm languages include C# and Java.

### Overview

The four main paradigms are described as follows:

**Imperative programming** describes computation in terms of statements that change the program state. Primary characteristics are assignments, procedures, data structures, control structures. Imperative programming can be seen as a direct abstraction of how most computers work, and many imperative languages are just abstractions of assembly language. Typical examples of imperative languages are C and Fortran.

**Object-oriented programming** describes computation in terms of objects described by attributes manipulated through methods. Primary characteristics are objects, classes, methods, encapsulation, polymorphism, inheritance. An example of a pure object-oriented languages is Smalltalk, while many other languages are either primarily designed for object-oriented programming (such as Java and C#) or have support for object-oriented programming (such as PHP and Perl).

**Declarative programming** describes computational logic without describing control flow, i.e. describing *what* a program does rather than *how* it does it. Many domain-specific languages such as SQL, HTML and CSS are declarative. Logic programming, such as Prolog, is a subset of declarative programming.

**Functional programming** describes computation in terms of mathematical functions and seeks to avoid program state and mutable data. Purely functional functions have no side effects, and the result is constant in relation to the parameters (e.g. $add(2, 4)$ always returns 6). An example of a purely function programming languages is Haskell. Other examples of languages designed for functional programming are Erlang, F# and Lisp, while it is possibly to apply functional programming concepts to many other languages.

While general-purpose languages, such as C# and Java, generally tend to lean towards the imperative and object-oriented paradigms, a domain-specific language, with very specific goals in design, may benefit from other paradigms, e.g. declarative programming.

Since our language is primarily meant for *declaring* board games, it would likely benefit from being a declarative programming language.

»»»»> fa40dc617f84b4ec4c756ecebc96bf5aae3763a6

## 2.9 Simulator

Considering the fact that board games consist of physical entities in the real world and rely purely on user-to-game and user-to-user interaction, we find it necessary to analyze how we can emulate this behavior in the most "realistic" way. To do this, we look at what a simulator is and could be, then what we can use it for, setting up some features an optimal simulator for our programming language would include, ending with a final definition of the simulator for our language.

So what is a simulator and what does it consist of? A simulator can be seen as a front end to an interpreter (or a compiler, though not as practical). It is the glue between the user and code execution. A user interacts with the simulator, which in turn interprets the user's input and does something with it, such as updating a graphical user interface or supplying some other kind of feedback.

Examples of simulators are seen in various different contexts, such as the Ruby[6] programming language's interactive shell *irb*, which is run from the command line and allows programmers to interact, experiment, and write code with immediate response, calling Ruby's interpreter upon every command entered. The *irb* keeps track of all current

code entered, allowing programmers to write an entire program in *irb*. Another example could be various different kinds of environmental simulators, such as physics simulators created by the University of Colorado at Boulder[9]. These simulators offer a computerized environment that allows changing of different factors within a simulated world, such as changing the pressure and gravity of an environment, providing instant feedback.

### Usage

We see the need for a simple simulator because board games consist of so much interactivity between the players and the board, that we need to mimic it. Nobody wants to sit and play noughts and crosses or chess in front of a terminal; that'd be both awkward and impractical. Therefore, we see the simulator playing a crucial role as the engine that drives the graphics and gameplay of a written game - in part being a front end to everything in the interpretation/compilation phases.

A board game designer could program his game in Garry and see it displayed with the current implementation fully working and playable on the screen in a matter of a few clicks. Another advantage with having such a simulator is that it can be used to prototype games before they physically need to be produced. Such a construction will allow quickly changing the game rules and board layout, etc. and support experimentation with different set-ups. This type of simulator could allow dynamically changing board game parameters, such as the board size, the amount of players, how the pieces behave, etc.

Another, more simple version of the simulator directed at the end users can be used to merely play the games. All they would have to do is open a game file in the simulator or set the simulator as the default program for game files written in our language. This is useful for games that don't necessarily need a physical version or when the game designer wants to test it with a broad group of people before putting it into production.

### Possible features

We decide that creating a set of potential features for a simulator will also be useful when it comes to designing the programming language itself, as these features can influence the syntax and semantics of the Garry language. Described below are some descriptions of possible features we have discussed and deem important for the simulator to offer.

**Interactive design** As a board game designer, it could be possible to quickly change pieces around and edit some things directly from the interface. This could influence the written code, creating a new game based off of the old one, much like the physics simulators mentioned previously. An alternative option to this would be to dynamically reload the file used as input if it is changed from an external source, allowing quick feedback if you're just editing a few lines in the game's source code.

**Loading pictures** Pieces and illustrations of various entities in the game can be automatically found and determined from their names definitions in a Garry file. This lets the designer think about writing a game and not how to load specific files from a directory and so on, easily influencing cluttered code.

**AI** As long as the code and game rules are well defined, an automatic AI could be implemented as a module in the simulator to simulate other players following the exact same set of rules, allowing the designer to test his entire game or parts of it without constantly needing other people. This could be very interesting, but unfortunately is out of the scope of this project.

**Multi-player** Multi-player support using the same computer or over a network. Each real player could take turns sitting at a physical computer, replacing non-existent players or computer-controlled players. As long as the simulator is implemented optimally, supporting multi-player games should be considerably simple, as the simulator needs to handle commands from a single player anyway. Scaling this up and handling multiple turns from multiple players shouldn't be too much of a challenge. A better, yet not always more practical solution is to allow players to play against each other across a network. Sending turn commands back and forth could be established via a simple protocol.

**Tracking moves** The simulator could offer a simple turn list displaying all the previous moves in the board game. Then it'd be possible to go back to a specific turn to "rewind" the game to a previous state.

These features could easily influence the syntax of our programming language. There could be specific reserved constructs to determine how the board and players are defined, making the simulator's job at displaying things easier.

### Definition of a simulator

We define a simulator as a package consisting of the language's interpreter/compiler and a GUI that is in direct contact with the users of our programming language. Whether these users are designers or players is irrelevant, as different versions of the simulator could easily be written and implemented. It can support many different features and could allow changes to be made as the user notices something that needs to be changed. The simulator sends commands to the interpreter/compiler and responds to the commands returned from it, such as updating a score, changing the position of a piece, or displaying an error message upon an attempting an illegal move.

An example of this could be that the user clicks and drags on a knight in an implementation of chess, moving it to another position on the game board. The simulator would send this behaviour to the interpreter or compiler (which recompiles), which checks it against the game's source code to see if the move itself is legal, and also any side-effects this move could have, such as eliminating an opposing player's piece.

We see spending time on writing a simulator useful because it links all the different stages together and will act as the final product containing all the other parts of the project. That said, it'd be ideal to separate the interpreter/compiler and simulator, allowing greater modularity if the interpreter/compiler is to be used in another implementation of a simulator or something entirely different.

Considering the fact that most board games are very visual and consist of different kinds of pieces placed at various different locations on a board, we conclude that we need a simulator. This simulator needs to be graphical and support all the elements a normal gaming session would, such as a board, pieces, rules for moving pieces, multiple players, and so on. Adding the ability to dynamically change programmatic features from the user interface is not rated as important, because this can simply already be done from the source code. It would help make testing and playing games as authentic as possible.

## 2.10 Requirements

This section presents a set of requirements for this project. The purpose of a requirements specification is to make sure that the final product does what is was intended to do and meets the specified requirements. It is used throughout the development phases and requirements are added as the project moves along and new challenges arise.

The requirements specification consists of three main points: functional requirements, non-functional requirements and solution goals. Functional requirements define what the final system should be able to do. Non-functional requirements define different constraints and boundaries for the entire project. Lastly, solution goals are overall requirements that help us define the correct solution to our problem statement [1].

**Functional requirements:**

- The programming language will be used to program board games **(Must have)**
- The programming language should as a minimum make it possible to implement chess and the special rules of chess **(Must have)**
- It must be possible to play the created board games in a graphical simulation **(Must have)**
- The programming language must be designed in such a way that it will be possible to keep track of the move history **(Must have)**
- It must be possible to play over a network in the simulation **(Wont have)**
- Players must have the possibility to undo a move in the simulation **(Could have)**

The list of requirements also have **non-functional requirements** which is split into two topics - performance limitations and project limitations.

**Performance limitations:**

- The programming language must be easy to learn and use by novice programmers **(Must have)**
- Programmers must be able to be able to implement board games with relatively few lines of code **(Must have)**
- The programming language must not be an extension of another programming language **(Must have)**
- The board games should as a minimum consist of two players **(Must have)**
- The source code of a single board game must be written in one file **(Must have)**
- The formal definition of the programming language must be described in Extended Backus-Naur Form (EBNF) **(Must have)**
- The programming language must either be compiled (by a compiler) or interpreted (by an interpreter) **(Must have)**

**Project limitations:**

- The programming language must be functional and operable no later than 29th of May 2013
- There group has approximately 20 hours per week to work on the project
- The project is limited by the group members' skills in the design development of programming languages
- The project (and hence programming language) must have a catchy name and logo

**Solution goals:**

- The programming language must make it easy and quick for programmers to develop board games which are within the scope of the defined problem statement **(Must have)**
- The board games must be playable on different operating systems **(Should have)**

**TODO:** Sum up here. What are the most important points?

## 2.11 Summary

Now that the analysis stage is complete. We are able to begin describing the design of our programming language. There has been defined a specific definition of board games, that should be implementable in Garry.

# Chapter 3

# Design

## 3.1 Grammar

### Notational Conventions

We use a variant of Extended Backus-Naur Form to express the context-free grammar of our programming language.

Each production rule assigns an expression of terminals, non-terminals and operations to a non-terminal. E.g. in the following example the non-terminal *decimal* is assigned the possible terminals of `"0"` up to and including `"9"`.

*decimal* → `"0"` | `"1"` | ··· | `"9"`

The following operations are used throughout this section:

| | |
|---|---|
| [ *pattern* ] | an optional pattern |
| {*pattern*} | zero or more repititions of pattern |
| ( *pattern* ) | a group |
| $pattern_1$ \| $pattern_2$ | a selection |
| `"0"` \| ··· \| `"9"` | a range of terminals |
| $pattern_1$ − $pattern_2$ | matched by $pattern_1$ but not by $pattern_2$ |
| $pattern_1$ $pattern_2$ | concatenation of $pattern_1$ and $pattern_2$ |
| `"test"` | a terminal |
| `"'"` | a terminal single quotation mark |
| `'"'` | a terminal double quotation mark |
| `"\"` | a terminal backslash character |

### Character Classes

| | | |
|---|---|---|
| *decimal* | → | `"0"` \| `"1"` \| ··· \| `"9"` |
| *lowercase* | → | `"a"` \| `"b"` \| ··· \| `"z"` |
| *uppercase* | → | `"A"` \| `"B"` \| ··· \| `"Z"` |
| *anycase* | → | *lowercase* \| *uppercase* |
| *quotebs* | → | `'"'` \| `"\"` |
| *unichar* | → | any unicode character |
| *strchar* | → | *unichar* − *quotebs* |

### Reserved Tokens

| | | |
|---|---|---|
| *keyword* | → | `"game"` \| `"piece"` \| `"this"` \| `"width"` \| `"height"` |
| | \| | `"title"` \| `"players"` \| `"turnOrder"` \| `"board"` |
| | \| | `"grid"` \| `"setup"` \| `"wall"` \| `"name"` \| `"possibleDrops"` |
| | \| | `"possibleMoves"` \| `"winCondition"` \| `"tieCondition"` |
| *operator* | → | `"and"` \| `"or"` \| `"not"` |
| *pattern_keyword* | → | `"friend"` \| `"foe"` \| `"this"` \| `"empty"` |
| *pattern_operator* | → | `"*"` \| `"?"` \| `"+"` \| `"!"` |

### Literals

| | | |
|---|---|---|
| *integer* | → | *deimal* {*decimal*} |
| *direction* | → | `"n"` \| `"s"` \| `"e"` \| `"w"` \| `"ne"` \| `"nw"` |
| | \| | `"se"` \| `"sw"` |
| *coordinate* | → | *upperccase* {*uppercase*} *decimal* {*decimal*} |
| *string* | → | `'"'` {*strchar* \| `"\"` *unichar*} `'"'` |

### Identifiers

| | | |
|---|---|---|
| *function* | → | *lowercase anycase* {*anycase*} |
| *identifier* | → | *uppercase* {*anycase*} |
| *variable* | → | `"$"` *anycase* {*anycase*} |

### Program Structure

| | | |
|---|---|---|
| *program* | → | {*function_def*} *game_decl* |
| *function_def* | → | `"define"` *function* `"["` {*variable*} `"]"` *expression* |
| *game_decl* | → | `"game"` *declaration_struct* |
| *declaration_struct* | → | `"{"` *declaration* {*declaration*} `"}"` |
| *declaration* | → | ( *keyword* \| *identifier* ) *structure* |
| *structure* | → | *declaration_struct* \| *expression* |

### Expressions

| | | |
|---|---|---|
| *expression* | → | *function_call* |
| | \| | *element operator expression* |
| | \| | *if_expr* |
| | \| | *lambda_expr* |
| | \| | *element* |
| | \| | `"not"` *expression* |
| *element* | → | `"("` *expression* `")"` |
| | \| | *variable* |
| | \| | *list* |
| | \| | *pattern* |
| | \| | *keyword* |
| | \| | *direction* |
| | \| | *coordinate* |
| | \| | *integer* |
| | \| | *string* |
| | \| | *identifier* |
| | \| | *function* |
| *function_call* | → | *function list* |
| *if_expr* | → | `"if"` *expression* `"then"` *expression* `"else"` *expression* |
| *lambda_expr* | → | `"#"` `"["` {*variable*} `"]"` `"=>"` *expression* |
| *list* | → | `"["` {*element*} `"]"` |

### PATTERNS

*pattern* → `"/"` *pattern_expr* {*pattern_expr*} `"/"`
*pattern_expr* → *pattern_val* [ `"*"` | `"?"` | `"+"` ]
*pattern_val* → *direction*
| *variable*
| *pattern_check*
| `"!"` *pattern_check*
| `"("` *pattern_expr* {*pattern_expr*} `")"` [ *integer* ]
*pattern_check* → `"friend"`
| `"foe"`
| `"empty"`
| `"this"`
| *identifier*

## 3.2 TYPES

The language should have support for the following types:

**Integer** a 32-bit signed integer.
**String** a UTF-8 encoded string.
**Direction** a directional value.
**Coordinate** a 2-dimensional vector consisting of an $x$-integer value and a $y$-integer value, representing a position in a grid.
**Function** a function reference (could be anonoymous, e.g. lambda expression).
**List** a list of values (can be empty).
**Action**

There is no *null*-type or *null*-value, since all expressions must have a value. This is also evident in the definition of the $if$-expression in section 3.1, in that all $if$-expressions must have the $else$-branch.

Dynamic typing.

## 3.3 FUNCTIONS

Many functions are made available to the programmer in GARRY.

## 3.4 SCOPING

Outline:

- What is scoping?
- Examples of static/lexical versus dynamic scoping
- Why do we want to use dynamic scoping
- What does that mean for GARRY?

# Chapter 4

# Implementation

## 4.1 Scanner

A scanners job is to analyse an input for lexical errors. The techniques required for doing a lexical analysis are more simple than the techniques required for syntactical analysis, and it is easier to optimize the lexical analysis phase if it is kept separated from the parser. We have decided to separate the lexical- and syntactical analysis like most other compilers or interpreters do. This section will cover the lexical analysis, which is performed by a scanner[7, p.189]

A scanner can be hard coded by hand or generated using existing tools, e.g. JFLex (for Java). When using tools like JFlex, you must define tokens to match input using regular expressions. If a programming language is complex, writing a scanner by hand can be very time consuming and error prone. On the other hand, learning how to use a tool like JFLex can also take a lot of time. Tools like Flex will often create faster scanners because they have incorporated many smart tricks to tweak the performance of the scanner generated[3, p.116]. We have been using JFLex in a course running beside this project, which is the reason why we have chosen to hand code the scanner for this project, to experience how that method works as well.

A finite automata can be used to recognise tokens specified using regular expressions. The automata can be table driven where each pair of a state and an input character matches an element in the table which points to the next state. A list of accept states tells in what states a string will be accepted and which token it will be accepted as. However, one can also code a DFA with explicit control, where the transition table that defines the DFA's actions is not declared explicit, but is incorporated as the control logic. The table driven DFA is often generated by tools that converts regular expression into the table used by the DFA. We have chosen to use the finite automata approach using the explicit control form, because most of us have no experience in writing a scanner and we think we will gain more from the explicit control form method rather than using a tool to do so. [3, p.94].

Our scanner takes a raw source code for a program written in GARRY as input. It validates the lexically correctness of a GARRY program. The scanner tries to find tokens existing in GARRY from the input. If an input is met which can not be recognized as a valid token, the source code is not a valid program in GARRY. **TODO:** Error handling? The strings which are converted to tokens are called lexemes. Many lexemes can be converted to the same type of token. Programs will typically contain many different identifiers, which in GARRY all will have a **ID**-token instantiated. The name of the identifier will then be saved as value belonging the particular token. Consider the example of this single line taken from a chess game written in GARRY:

```
1 Black{ Pawn [A7 B7 C7 D7 E7 F7 G7 H7] }
```

The result of analysing the input can be seen in table 4.1. Tokens are needed for abstraction. When the parser later on will determine if the code respects the grammar of GARRY, it is useful to have these abstractions. It makes it

| Lexemes | Tokens |
|---|---|
| Black | IDENTIFIER (Black) |
| { | LBRACE |
| Pawn | IDENTIFIER (Pawn) |
| [ | LBRACKET |
| A7 | COORD_LIT (A7) |
| B7 | COORD_LIT (B7) |
| C7 | COORD_LIT (C7) |
| D7 | COORD_LIT (D7) |
| E7 | COORD_LIT (E7) |
| F7 | COORD_LIT (F7) |
| G7 | COORD_LIT (G7) |
| H7 | COORD_LIT (H7) |
| ] | RBRACKET |
| } | RBRACE |

Figure 4.1: *Analysing an input stream for lexemes and tokens*

possible to describe that a list of **COORD_LIT**-token's can be encapsulated between the characters "[ ]" without having to list all possible coordinate literals, which in fact are an infinite set, since the grammar of GARRY allows proceeding in both dimensions after *Z9*, namely *Z10* and *AA9*. However, like an identifier, the value of other token types, for instance coordinate literal is still kept since that information will be needed later for the subsequent parts of the interpreter.

Our scanner contains 2 classes, **Scanner** and **Token**. **Token** contains an enum named **Type** that enumerates all the types of tokens in GARRY. When a lexeme is found in the input stream, the scanner analyses which token type it belongs to. A new token is then instantiated and returned by the scanner. The constructor for **Token** takes the arguments (**Token.Type**, *line*, *offset*). The *line* and *offset* represent where in the source code the lexeme of any token where found. This is essential if an error is found, since we can then inform a programmer where in his source code he should look for the error.

When an input is to be analysed, the scanner looks at the first symbol of the input and determines which subfunction to jump to. This is where the explicit control of the DFA is seen. If we had used a table driven automata, the program would just update a variable **currentState**. Instead this variable exists implicit as the call stack showing which subfunctions we have jumped into. In example 4.2, you can clearly see many functions with the name **isSomething()**, which simply returns if the next symbol in the input stream is "Something". **isWhitespace()** returns true if the next input is a whitespace. While the condition is true, **pop()** dequeues the next symbol. Therefore, the while loop with **isWhitespace()** removes all initial white spaces before a next token is found. After that, if the scanner has reached the end of the input stream, it returns a **EOF**-token. For all **isSomething()** functions, except the **isWhitespace()** function, a token will be returned based on some evaluations the subfunction is responsible for. For example, if the first symbol of a lexeme is an upper-case character, the function **scanUppercase()** is responsible for determining whether the lexeme is an **identifier**-token or a **direction**-token, because they are the only tokens starting with an upper-case character in GARRY. Two variables, **int offset** and **int line** keeps track of where in the source file the next input character is taken from. The function **pop()** will pop the first character from the input stream and assign the value of the new first character to the variable **nextChar**. The **isSomething()** functions uses that **nextChar** to check what kind of character the next one is. The **pop()** function will additionally increment **offset** by one. If the next input symbol is a whitespace, it assigns zero to **offset** and increments **line** by one. If an input symbol is met which was not expected, an error is outputted. **TODO:** What to do about errors here?

```
public Token scan() throws Exception {
  while (isWhitespace()) {
      pop();
    }
    if (isEof()) {
      return new Token(Type.EOF, line, offset);
    }
    if (isDigit()) {
      return scanNumeric();
    }
    if (isUppercase()) {
      return scanUppercase();
    }
    if (isOperator()) {
      return scanOperator();
    }
    if (isLowercase()){
      return scanKeyword();
    }
    if (peek() == '"'){
      return scanString();
    }
    if (peek() == '\$'){
      return scanVar();
    }
    throw new Exception("Could not find a valid token starting with char" + peek());
  }
}
```

Figure 4.2: *The scan() function from the* GARRY *scanner.*

CHAPTER 5

# EVALUATION

CHAPTER 6

# CONCLUSION

# BIBLIOGRAPHY


[1] Stig Andersen. Den gode kravspecifikation. page 3, May 2006. http://www.infoark.dk/article.php?alias=kravspec.

[2] Chess.com. Learn to play chess. http://www.chess.com/learn-how-to-play-chess. (2013-02-13).

[3] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Pearson, global edition edition, 2009.

[4] Ulrich Neumann J.P.Lewis. Performance of java versus c++. http://scribblethink.org/Computer/javaCbenchmark.html. Seen 11/3/13.

[5] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Learning, 2nd edition edition, 2006. ISBN 0-534-95097-3.

[6] RubyIdentity. Ruby - a programmer's best friend. http://www.ruby-lang.org/en/. Seen 8/3/13.

[7] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, international, 10th edition edition, 2013.

[8] Aalborg Universitet. Bacheloruddannelsen i Datalogi. http://www.sict.aau.dk/digitalAssets/50/50637_datalogi-bachelor.pdf, June 2011.

[9] University of Colorado at Boulder. Interactive Simulators. http://phet.colorado.edu/en/simulations/category/physics/index. See timestamp for visited date.

[10] wiseGeeks. What is a board game? http://www.wisegeek.com/what-is-a-board-game.htm. (2013-02-12).

# APPENDIX


# APPENDIX