Aalborg Universiy

Computer Science

Junta

Generic Board Game Programming

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:

Junta - generic board game programming

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

Abstract:

Preface

This project describes the development of a domain-specific programming language that can be used to describe board games. The specific topic was to design, define, and implement a programing language for generic game playing.

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.[14, p. 22] We will discuss different paradigms in section 2.3.

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.[14, p. 22]

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

Contents

CHAPTER 1

INTRODUCTION

1.1 A NEW PROGRAMMING LANGUAGE?

So, are we really going to create a new programming language? At this very moment, hundreds maybe thousands of new programming languages are being created. A Wikipedia article about the programming languages that exists, currently contains a list of more than 660 different programming languages [17], and a lot more languages exists, since this is only a list of the more well-known languages. So why is yet another programming language needed when there exists so many programming languages already, and when programming languages like Java, C# and other general purpose programming languages basically make it possible to program anything one would like. The answer lies within the powers of domain-specific programming languages - a language designed to express solutions to problems in a specific domain [3].

What we intend to do in this project is develop a programming language which can describe board games. That is designing it, defining it and implementing it. We choose to develop a programming language for the domain in which the programming of board games lies within, because of our personal interest in board games. Board games are first of all fun and entertaining. They can also be rich on learning opportunities [10] and they can give a certain satisfaction [6]. In fact, board games can be almost anything as long as the creator of the game has the right amount of creativity and the building blocks needed to create the game. These are some of the reasons why it would be practical with a programming language which makes it easy to program board games. Yes, it is possible to program board games in Java, C# and the other general purpose programming languages, but is it easy? Definetly not as easy as in a well crafted programming language specifically designed to program board games in. If a language purely concentrates on allowing the programmers to use it to express how the game works, without having to reimplement everything from scratch as he would in a general purpose programming language, it would be quicker and easier to develop shorter and more precise programs.

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 allow the programmers to, relatively quickly, explore new ideas for a board game with a simple implementation. The programmer could then efficiently modify the code according to a new rule or idea that suddenly came up, without having to make major changes. If the language also took multiple platforms into account, it would open up the possibility to run the same game across multiple devices. So there are definitly many advantages of creating a new domain-specific programming language.

Now the question is, how to develop a new domain-specific programming language in which one can program board games? First we will have to have to do some research about board games and the elements and components they

consist of. We will have to know about the different programming paradigms there exists, we will have to know about compilers and/or interpreters and many other things. All of these things are included in chapter 2

Chapter 2

Analysis

In the following chapter we analyse some different board games, namely Kalah and Chess. This is done to gain a better understanding of which elements and components they contain, which will be usefull knowledge when designing our programming language. The analysis of the games can be found in section 2.2. Furthermore, we investigate programming paradigms in section 2.3. We need to know about the existing paradigms to be able to make a good decision, on which paradigm(s) our own language should be based on. In the chapter we also analyse the phases of compilation and interpretation. We give an overview of their phases, which is seen in section 2.4, and after that we dig deeper into the details of the parsing phase where we describe what a context free grammar is, in section 2.5, and we look at the advantages and disadvantages of different kind of parsers and different ways of constructing parsers in section 2.6. In the end of the chapter a list of requirements to our programming language is presented together with a problem statements. These are the results of the chapter and will work as the foundation for the further work of the project.

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 [19]:

> "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 likely have to be a general purpose programming language. If a programming language is aimed to make the programming of only a specific kind of board games easier, there might be many things that can be expressed easy in that language compared to how it would have been done in existing general purpose programming languages.

To define what elements that is to be included in JUNTA, we think it is essential to look at some existing board games. For that reason we have analysed two well known board games. We have investigated the game elements that might be clumsy or not straightforward to implement in common general purpose programming languages. After the analysis of the games, a list of elements is served that respects all of the elements from the games. The aim of JUNTA is to ease the programming of these game elements

2.2 CHESS AND KALAH

In the following sections we will analyse the two games: Chess and Kalah. The reason for picking these games is because they are among those we have biggest personal interest in, and we think they contain some fundamental board game elemenents which we need to know about to gain a better understanding of which features are needed in a board game programming language. We are going to focus on the components of the games e.g. the pieces, the board, the squares, etc.

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 8×8 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 other games e.g. 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 chess from a lot of other common board games where all pieces have the same abilities, like Naughts and Crosses, Mancala, Ludo, Backgammon.

Cut to the bone Chess goes as follow: When a game starts the pieces are in their 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 condition is obtained when the king piece is in a position to be captured and cannot escape from capture in the next move. [4]. Therefore it's nessecary to look one move ahead to control if the checkmate condition is optained.

Special moves

In chess there are numerus special moves which doesn't follow the normal pattern of chess. Earlier we mentioned that a pawn can 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 that it can only move one square forward or one square vertically 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 [4]. There exits two more special moves which are called "En Passant´´ and "Promotion´´. These are not going to described here, but information about them can be found in [4]. So what is the problem with these special moves? The problem is the fact that they don't follow the regular pattern of the game and this has to be taken into consideration when designing JUNTA.

From the above analysis here is a list of interesting game elements we found in chess:

- Pieces has different movement abilities.
- A squared 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.
- Special moves like "Castling", "Promotion" and "En Passant.

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

- Constraints that disallows a piece to move if some condition is true after the move has been made (a move that sets your king in check).
- A piece can be "captured" by another piece, which causes the piece to be removed from the board.

KALAH

Kalah is like chess a turn-based game of two opponent players. The Kalah pieces, called seeds, are very different from the Chess pieces. They do not have specific moves but rather functions, as their name also suggest, as seeds. The board is not like the Chess board either. It consists of 14 squares, sometimes refered to as houses [16], with two of the houses separating themselves from the rest by beeing the houses or bases of each of the players. Further more each player has six houses belonging to him/her. See figure 2.2. Each house (including the players houses) can contain an arbitrary number of pieces/seeds, unlike in chess where the squares can only contain one piece.

cut to the bone Kalah goes as follow: When the game starts each of the 12 houses contains 4 seeds (in some versions of the game each house cointain 5 or 6 seeds) and the player bases are empty. Now the players shifts in turn to pick up piles of seeds and dealing them out to the 12 houses and his own base. The dealing of seeds works by a player picking up a pile of seeds from one of his six houses, and dropping one seed down in each of the following houses moving counter-clockwise. If, when dealing out seeds, he lands in a house belonging to himself (not including his own base), which is not empty and which is not belonging to the opponent, he can pick up the pile of seeds in the house and start dealing these out. The turn shifts once a player, when dealing out seeds, lands in an empty house or in one of the opponent's houses. For a more detailed description of the rules, we refere to [16]. The game is over, once one of the players six houses are empty and the winner is who ever has the most seeds in his/her base.

Special moves

Like in Chess there are some special moves in Kalah which doesn't follow the regular pattern of the Kalah game. We are not going to describe them here, but they will be present in the list of interesting game elements and can be found in a detailed description in [16].

Here is a list of some interesting game elements we found in Kalah. For simplicity we are going to refere to houses and bases as squares again and refere to seeds as pieces again:

- Squares can contain an arbitrary number of pieces
- Making a move can be considered as simply choosing a square
- The number of pieces on a square determines how long a move you can make
- A turn may contain more than one move
    - If the last piece is dropped in a not-empty square, the player can make another move.
- A square can belong to a player.
- Squares can be related to other squares.

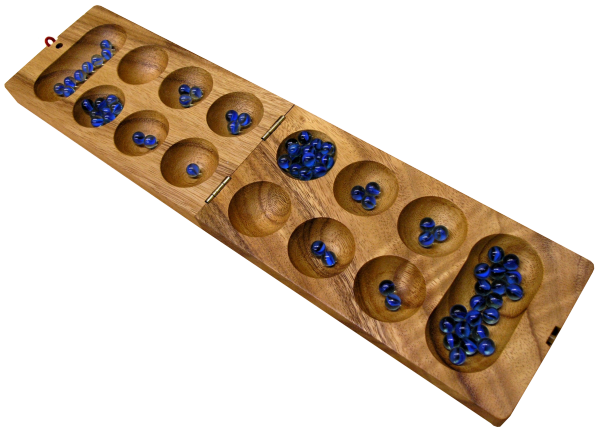

Figure 2.2: *The board game Kalah. There are six squares on each side belonging to each player, and there are two square in each end of the board, which represents the bases of the players.*

    - You place pieces on squares counter-clockwise.
    - If you place the last piece on an empty square, the square across the board belonging to the opponent is emptied over to your own square.
- An end game condition - when all of the square belonging to a player are empty
- A winning condition - the player with the most pieces in his/her square when the end game condition has been met
- Only one type of piece

Summarised interesting game elements

From looking at the board games Chess and Kalah, many different game elements have been recognised. For a programming language that allows theses two games to be implemented, all of the game elements must be possible to be designed in the programming language. A summarised list of the game elements has been created. In the list some of the elements from the above analysis have been combined and described by a more general element. Also some elements from the above analysis have been split up and been formulated into more detailed ones and some new elements has been found which has also been added to the list. The resulting list looks as follow:

- A game has an initial setup.
- A game can be a tie if the game is in one or more specific states.
- A player can win if the game is in one or more specific states where the winning conditions are met.
- A piece can be either on the board or off the board (e.g. If a piece has been captured in chess).
    - If a piece is on the board it is positioned on one specific square.
    - A piece off the board may be put back on the board. (This happens in the promotion move in chess).
- A game can contain of one or more types of pieces.
- A piece can belong to a player.
- At any given time, just one player has the turn.
- A player's turn can consist of more than one move.
- If a piece is owned by a player, only that player can use it's moves.
- A game has a single board.
- A board contains of squares in a two-dimensional grid.

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.

2.3 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. It is essential for us to be aware of these different paradigms, since it may help us to find a good style of programming for the design of Junta.

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.

2.4 Compiler overview

The following section presents a brief overview of the phases of translators (compilers and interpreters). The typical translator takes as input some given source code written in a language with a high level of abstraction and traslates it into a language with lower abstraction e.g. machine code which can be executed directly by a computer [11, p. 44]. Some translators work differently though. They translate the source code into another high-level language or into machine code for virtual machines, which can provide portability. The translation process is typically not a simple task, therefore it is often split into different phases, which is shown in figure 2.3. The process can be split into more or less phases though, depending on how detailed one wished to describe the process. In this section we describe the following phases: the lexical analysis, the syntax analysis, the semantic analysis, the code generation and the interpretation.

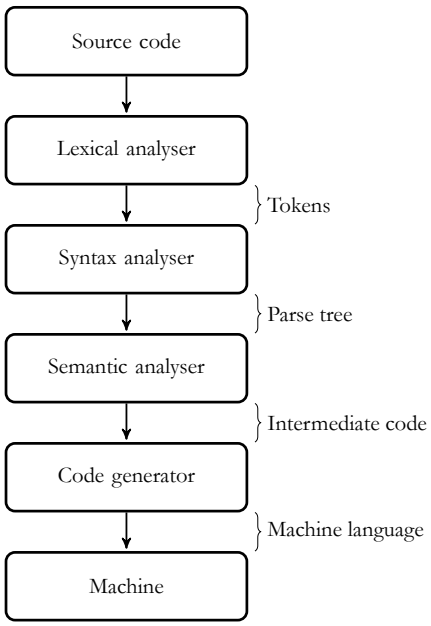

Figure 2.3: *The different phases of a compiler. Based on Sebesta et al.[11] p. 46, Figure 1.3*

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[11, 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	**Tokens**
$a	var(a)
=	assign
3	int(3)
$b	var(b)
+	plus
4	int(4)
$a	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[11, 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[11, 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.

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

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[11, 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* → *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 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[5, 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[5, p. 551]. Some of the problems associated with the pipelined execution is solved by move apart instructions that will interlock[5, p. 552].

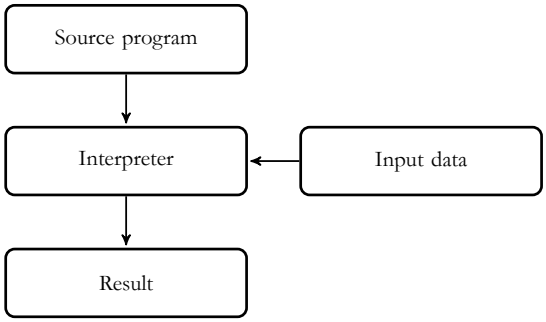

Figure 2.4: *The different phases of an interpreter. Based on Sebesta et al.[11] p. 48, Figure 1.4*

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 [11, 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[11, p. 50].

2.5 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, Σ, R, S):[8, p. 100]

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

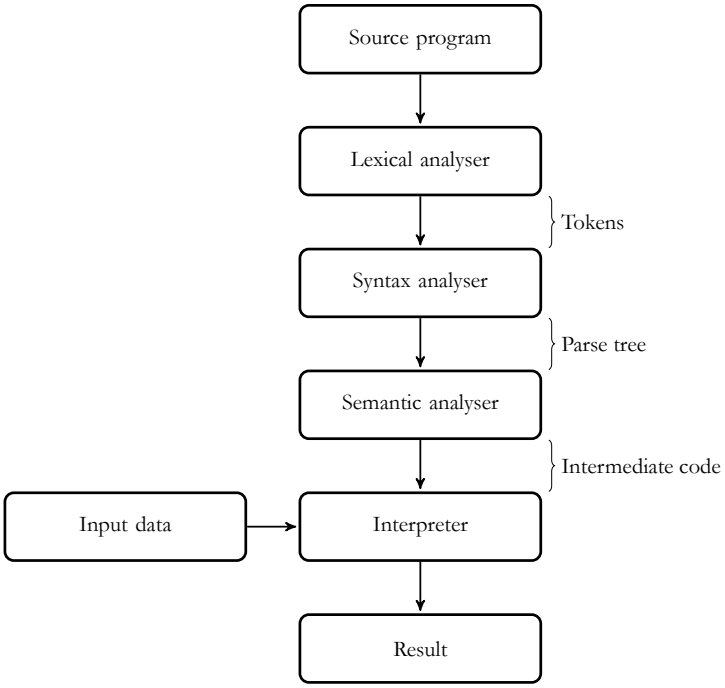

Figure 2.5: *The different phases of a hybrid implementation systems. Based on Sebesta et al.[11] p. 49, Figure 1.5.*

- 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**:

$$\begin{array}{lcl} A & \rightarrow & aAb \\ & | & B \\ B & \rightarrow & c \end{array}$$

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.[8, 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.

2.6 Parsers

In the following section we are going to compare the LL parsers and the LR parsers, by analysing the advantages and the disadvantages of each. Further more we are going to look at different methods for making parsers, more specifically we are going to analyse the pros and cons against writing the parser by hand versus using parser generator tools to generate it. 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. There exists two main approaches to parsing, which have very different ways of dealing with the parsing process and which we are going to describe briefly, that is top-down parsing and bottom-up parsing. From the top-down parsing approach derives the family of LL(k) parsers (the k defines the number of lookahead tokens), where the LL(1) parser is the more popular and widely used one. From the bottom-up approach derives the family of LR(k) parser, which includes the LALR(1) parser and several others.

Top-down parsing and bottom-up parsing

In the top-down 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 bottom-up 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 as mentioned above - hence the name top-down parsing. While LR-parser builds the parse tree from the branches up to the root - hence bottom-up.

The parsers in action

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 continuously called until the entire input string has been eaten and thereby has been matched. An example of a LL(1) parser in action can be seen in table 2.2. In the example the parser is based on the simple grammar in table

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & T + E \\ & | & T \\ T & \rightarrow & int \end{array}$$

The LR-parsers also has two actions: the shift action and the reduce action. The shift action adds the next input symbol of the input string 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 continuously called until the input string is reduced to the start symbol. [12]. An example of a LR(2)-parser in action is illustrated in table 2.3.

Comparison of the parsers

Compared to the LL-parsers the LR-parsers are more complex and in general harder to write. The result of this however, is the fact that the LR-parsers are more powerful than the LL-parsers, powerful in the sense that they

	The process		
Step	Production	Input	Action
1	S	$int + int$	Predict $S \rightarrow E$
2	E	$int + int$	Predict $E \rightarrow T + E$
3	$T + E$	$int + int$	Predict $T \rightarrow int$
4	$int + E$	$int + int$	Match int
5	$+E$	$+int$	Match $+$
6	E	int	Predict $E \rightarrow T$
7	T	int	Predict $T \rightarrow int$
8	int	int	Match int
			Accept

Table 2.2: *A LL(1) parser seen in action parsing the string "int + int".*

	The process		
Step	Production	Input	Action
1		$int + int$	Shift
2	int	$+int$	Reduce $T \rightarrow int$
3	T	$+int$	Shift
4	$T+$	int	Shift
5	$T + int$		Reduce $T \rightarrow int$
6	$T + T$		Reduce $E \rightarrow T$
7	$T + E$		Reduce $E \rightarrow T + E$
8	E		Reduce $S \rightarrow E$
	S		Accept

Table 2.3: *A LR(2) parser seen in action parsing the string "int + int".*

accept a bigger variety of grammars and they parse faster. For instance the LR-parsers can handle grammars with left-recursion, but LL-parsers can't.

Constructing a parser

The principle of generating parsers is very systematic and therefore there are different automated tools to generate parsers for a specific grammar that meets some standards. A grammar must for instance not be ambiguous otherwise the tools cannot make a distinct parser for the grammar. We start by taking a look at the constructing of a handwritten parser. Then we take a look a two different parser generators that produce different parsers. Finally we sum up the pros and cons of the analysis on handwritten and generated parsers.

Handwritten parsers

Why would you write a parser on your own when you have automated tools for this job? If we were to construct our own handwritten parser, and not use the tools alreadt built for this, it would be so that we would gain a greater understanding of how these parsers work. How are they constructed? What kind of errors could occur when trying to develop a parser? One of the best ways to learn is to fail - learn from the mistakes and correct them. But this is also a pain in the neck if a lot of errors are popping up and the person trying to get the job done does not have the expertise to resolve the errors and find a solution.

So we would gain experience and probably learn a lot by writing our own parser - but what are the pitfalls of taking on this task? First of all we could be stumbling upon many errors in the code. These errors must be solved before the parser can be finished. Therefore the construction of the parser will be time-consuming. So we will be gaining knowledge about the process but it will take a lot of time compared to an automatic generator.

Programming languages evolve and their grammar can change. When this is the case the parsers must be maintained so that they still output the correct result. When we have generated a parser by hand this task will be time-consuming

because we must search through the code of the parser and tweak it so it will be correct again. Whenever we work on tweaking existing code we are most likely going to run into new errors that need to be resolved.

This brings us to the topic of how reliable our handwritten parsers are compared to the generated parsers. So far we have discussed that the produced code for a handwritten parser will be error-prone so this will naturally bring us to conclude that this code must be less reliable than the automated generators produced code. This is a big con because we have to be able to rely on the output of the parser.

Generated parsers

There are quite a few automated parser generaters (compiler compilers). To construct af parser with a generator the developer must input the grammar into the generator, and it will output a parser for that specific grammar. This can be a bit different from software to software but the grammar is often inputtet as Extended Backus Naur Form (EBNF) and the output parser will be produced in the language the generator is meant to output. We take a short look at SableCC and JavaCC in the following two section that both produce Java source code.

What differences are there between a handwritten and a generated parser? We need a grammar in both methods so what is the advantages of a generated parser? First of all, it takes less time to construct the parser because once we have a grammar we can input it in the generator and it will automatically generate the parser for us. Before we can use the software we have to figure out how to use it but this is not a complicated process. For every software there is some kind of tutorial on how to use the software.

The software that generate the parsers have been under development for quite a while and therefore the developers are using efficient algorithms to implement the parsers. This means that the parser we construct by hand will not be as fast and reliable as the ones generated by the software - unless the programmer is very experienced with a wide knowledge base about this subject. So the generated parser is most-likely more efficient.

Maintenance of the parser is also much easier because everything is machine-produced and can easily be changed to correspond witht he new grammar if the grammar has been changed.

There are different methods for constructing parsers. We have top-down parsers, where the parse trees are built from the root (the top) to the bottom, and bottom-up parsers, where the parse trees are built from the bottom to the root. Different grammars have different limitations and the different types of parsers work on specific grammars. We will shortly discuss this in the following sections.

SableCC, a bottom-up parser

SableCC is a bottom-up parser generator that generates LALR(1) parsers. This generator runs on the Java-platform and produces object-oriented code with clearly seperated machine-generated code and handwritten code. This contributes to the simplification and ease of maintaining the code.[13, pp. 11]

The following is a list of advantages that LR parsers have:[11, pp. 193] **TODO:** kontroller sidetal! afsnit 4.5.3

- LR parsers can be built for all programming languages
- They can detect syntax errors as soon as it is possible in a left-to-right scan
- The LR class is a proper superset of the class parsable by LL parsers
    - Many left-recursive grammars are LR, but none are LL.

The only disadvantage a LR parser has is that it is verey difficult to produce by hand. We have the automated generators to solve this disadvantage.[11, pp. 193]

JavaCC, a top-down parser

JavaCC is a top-down parser generator that generetes LL(k) parsers. As the name of the parser generator it also produces the output code as Java source code.[18]

The LL class of grammars cannot contain left-recursion like some LR grammars can. This means that the LL class of grammars is less powerful than LR parsers, because LR grammars can be made for any LL grammar but not any LL grammar can be made for any LR grammar because LR is a superset of LL.

Summary

By reading the above section about handwritten parser we can conclude the following advantages by handwriting a parser:

- Gain experience in constructing parsers
- Gain a better understanding of how parsers work

We will be gaining a lot of experience by writing a parser by hand and solving the problems that arise along the way. But there are quite a few disadvantages accompanied with handwriting a parser, and they are as follows.

- Can be error-prone
- Can be time-consuming to construct
- Time-consuming to maintain
- Less reliable than generated parsers
- Slower than generated parsers

We've summed up the advantages and disadvantages of handwritten parsers. Know we take a short look at the advantages of generated parsers. By reading the above section about the automatically generated parsers we can conclude the following advantages:

- Efficient
- Reliable
- Fast
- Easy to maintain

These parsers will most-likely be more efficient and faster than handwritten parsers because they include efficient algorithms developed and maintained throughout the lifetime of the software. This also makes them more reliable than handwritten parsers because there will be much less errors in the process of constructing the parser.

This brings us towards a conclusion on handwritten and generated parsers. It is very clear that there are more benefits in using an automated generator to construct a parser. It is quite obvious that it will be much easier to reach our goal of constructing a parser. But we believe that it is very important that we try to gain experience and therefore we will be both handwriting a parser and using a generator as well.

2.7 Compiler and interpreter

Along with the design of the programming language JUNTA, we also want to make it possible for programmers who write games in JUNTA 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 file 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 at runtime compiles intermediate code into native machine code. This process of course adds an overhead, however the speed differences are not that great anymore[7].

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 2.7.

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 one wouldn't be able to notice the difference between a compiled and a interpreted version. One place where speed does become important is in an artificial intelligence (AI), which would be used to create an virtual opponent controlled by the computer. The virtual opponent becomes harder to beat the more turns it can look ahead, however the computational 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 JUNTA. 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.

Summary

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.

2.8 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 analyse how we can emulate this behaviour in the most "realistic" way. To do this, we look at what a simulator is and could be, what we can use it for, and set 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[9] 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[15]. 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 JUNTA 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 JUNTA 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 JUNTA 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.

Chapter 3

Requirements

In the previous chapter we have presented our problem analysis. This chapter presents the requirements for our programming language which we have reach through our analysis.

3.1 Requirements

This section presents a set of requirements for this project which has been structured using a method published by Stig Andersen[1]. 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 [2].

Functional requirements:

- The programming language will be used to program board games
  - As a minimum, it must be possible to implement chess and the special rules of chess
- It must be possible to define what pieces the game consists of
- It must be possible to define which pieces a specific player is able to move to
- It must be possible to define the possible squares a piece can be moved to
- It must be possible to represent list structures
  - It must be possible to perform list unions
  - It must be possible to perform list intersections
- It must be possible to use the language's built-in functions to do the following:
  - determine if a square is empty
  - determine if a square is occupied, and by who
  - check a condition for all objects in a collection
  - find all squares that match a specific pattern
  - combine lists
  - perform a lambda expression on each element in a list

    - move a piece to a square
    - capture the old piece on a specific square while moving a new piece to the same square
    - return which players turn it is
    - check if the current move about to be made for a piece is the first move made by that piece
- It must be possible to determine which legal moves a player has
- It must be possible to define winning conditions
- It must be possible to define draw conditions
- It must be possible to perform integer arithmetic
    - Addition, subtraction, multiplication, and division
    - The programming language must have boolean operators
    - The programming language must have comparison operators
- It must be possible to perform string concatenation
- No function nor expression may produce side effects
- There must be an action type that handles game state changes
- It must be possible to create lambda expressions
- It must be possible to declare functions
- It must be possible to reference functions
    - Functions must be first-class citizens
    - It must be possible to call functions
- It must be possible to non-destructively assign any value to variables
- The created board games must be playable in a graphical simulator
- The simulator must be able to remember move history
    - It must be possible to undo/redo moves
    - It must be possible to save the move history
        - It must be possible to start a game from a saved move history
    - It must be possible to play over a network

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

Performance limitations:

- It must be easy to learn how to program in the programming language
- The programmer must be able to implement board games with relatively few lines of code
- The programming language must not be an extension of another programming language
- The board games should as a minimum consist of two players
- The source code of a single board game must be written in one file
- The formal definition of the programming language must be described in Extended Backus-Naur Form (EBNF)
- The programming language will be interpreted (not compiled)

Project limitations:

- The programming language must be functional and operable no later than 29th of May 2013
- The group has approximately 20 hours per week to work on the project
- The project is limited by the group members' skill in the design and development of programming languages
- The project (and hence the 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
- The board games must be playable on different operating systems

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

Chapter 4

Design

```
// An implementation of the traditional
// Noughts and Crosses game
game {
  title "Noughts and Crosses"
  players [Noughts Crosses]
  turnOrder [Crosses Noughts]
  board {
    grid {
      width 3
      height 3
    }
  }
  piece {
    name XOPieces
    possibleDrops (emptySquares[board])
  }
  winCondition (
    (notEmpty[(findSquares[/friend n friend n friend/])])
      or (notEmpty[(findSquares[/friend e friend e friend/])])
      or (notEmpty[(findSquares[/friend nw friend nw friend/])])
      or (notEmpty[(findSquares[/friend ne friend ne friend/])])
  )
  tieCondition (
    isFull[board]
  )
}
```

4.1 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"`.

$$decimal \rightarrow \texttt{"0"} \mid \texttt{"1"} \mid \cdots \mid \texttt{"9"}$$

The following operations are used throughout this section:

$[\, pattern \,]$	an optional pattern
$\{pattern\}$	zero or more repititions of pattern
$(\, pattern \,)$	a group
$pattern_1 \mid pattern_2$	a selection
$\texttt{"0"} \mid \cdots \mid \texttt{"9"}$	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

$$\begin{aligned}
decimal &\rightarrow \texttt{"0"} \mid \texttt{"1"} \mid \cdots \mid \texttt{"9"} \\
lowercase &\rightarrow \texttt{"a"} \mid \texttt{"b"} \mid \cdots \mid \texttt{"z"} \\
uppercase &\rightarrow \texttt{"A"} \mid \texttt{"B"} \mid \cdots \mid \texttt{"Z"} \\
anycase &\rightarrow lowercase \mid uppercase \\
quotebs &\rightarrow \texttt{'"'} \mid \texttt{"\"} \\
unichar &\rightarrow \text{any unicode character} \\
strchar &\rightarrow unichar \; - \; quotebs
\end{aligned}$$

Reserved Tokens

$$\begin{aligned}
keyword &\rightarrow \texttt{"piece"} \mid \texttt{"width"} \mid \texttt{"height"} \\
&\mid \texttt{"title"} \mid \texttt{"players"} \mid \texttt{"turnOrder"} \mid \texttt{"board"} \\
&\mid \texttt{"grid"} \mid \texttt{"setup"} \mid \texttt{"wall"} \mid \texttt{"name"} \mid \texttt{"possibleDrops"} \\
&\mid \texttt{"possibleMoves"} \mid \texttt{"winCondition"} \mid \texttt{"tieCondition"} \\
shared_operator &\rightarrow \texttt{"*"} \mid \texttt{"+"} \\
normal_operator &\rightarrow \texttt{"-"} \mid \texttt{"!="} \mid \texttt{"=="} \mid \texttt{"<="} \mid \texttt{">="} \mid \texttt{"<"} \mid \texttt{">"} \\
operator &\rightarrow shared_operator \mid normal_operator \mid \texttt{"/"} \\
pattern_operator &\rightarrow \texttt{"?"} \\
pattern_not &\rightarrow \texttt{"!"} \\
pattern_or &\rightarrow \texttt{"|"} \\
pattern_keyword &\rightarrow \texttt{"friend"} \mid \texttt{"foe"} \mid \texttt{"empty"}
\end{aligned}$$

Literals

$$\begin{aligned}
integer &\rightarrow decimal \; \{decimal\} \\
direction &\rightarrow \texttt{"n"} \mid \texttt{"s"} \mid \texttt{"e"} \mid \texttt{"w"} \mid \texttt{"ne"} \mid \texttt{"nw"} \\
&\mid \texttt{"se"} \mid \texttt{"sw"} \\
coordinate &\rightarrow uppercase \; \{uppercase\} \; \; decimal \; \{decimal\} \\
string &\rightarrow \texttt{'"'} \; \{strchar \mid \texttt{"\"} \; unichar\} \; \texttt{'"'}
\end{aligned}$$

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	→	*element operator expression*
	|	*assignment*
	|	*if_expr*
	|	*lambda_expr*
	|	*element* [*list*]
	|	`"not"` *expression*
element	→	`"("` *expression* `")"`
	|	*variable*
	|	*list*
	|	`"/"` *pattern* `"/"`
	|	*keyword*
	|	`"this"`
	|	*direction*
	|	*coordinate*
	|	*integer*
	|	*string*
	|	*identifier*
	|	*function*
assignment	→	`"let"` *variable* `"="` *expression* {`","` *variable* `"="` *expression*} `"in"` *expression*
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* `"|"` *pattern_expr*
pattern_val	→	*direction*
	|	*variable*
	|	*pattern_check*
	|	`"!"` *pattern_check*
	|	`"("` *pattern* `")"` [*integer*]
pattern_check	→	*pattern_keyword*
	|	`"this"`
	|	*identifier*

4.2 Types

JUNTA has support for the following types: Integer, String, Direction, Coordinate, Function, Pattern, List and 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 4.1, in that all if-expressions must have the $else$-branch.

Integer

The integer-type in JUNTA represents a 32-bit integer.

An integer-value can be created using an integer-literal, such as `2155` or `0`.

String

The string-type represents a UTF-8 encoded string.

A string-value is created using a string-literal, such as `"Hello, World!"` or `""`.

Direction

The direction-type represents a direction on a game board. It works like a vector in the sense that they can be combined to compute new directions. The basic directions are `n`, `e`, `s` and `w` (north, east, south and west). On a 2-dimensional grid (such as for chess) north is up, east is right, south is down and west is left.

The directions `ne`, `nw`, `se` and `sw` are also available, although these could also be produced by combining the basic directions (e.g. `n + e = ne`). An example of a direction combination is the expression `n + n + e` which produces the direction shown in figure 4.2. This direction could also be produced by `n + ne` or `ne + n`.

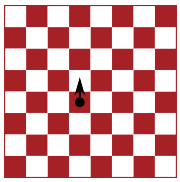

Figure 4.1: *The `n`-direction.*

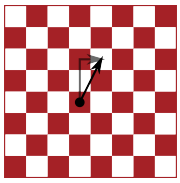

Figure 4.2: *The `n + n + e`-direction.*

Coordinate

This type represents a position in a grid, i.e. on the game board. It is created with a coordinate-literal, e.g. `A1` or `AH23`. The first part (the alphabetical part) represents the column (or x-value), i.e. `A` means column 1 and `AH` means column $1 \cdot 26 + 8 = 34$. The second part (the numeric part) represents the row (or y-value).

Function

Functions in JUNTA are first-class citizens, meaning that they can be used as any other value. A function name without a list of arguments results in a reference to that function. Function references can be passed as arguments to other functions or as a return value.

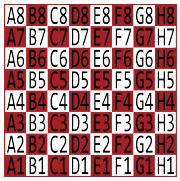

Figure 4.3: *All coordinates on an* 8×8 *board.*

Another way to create function references, is to use anonymous functions in the form of lambda expressions. A lambda expression is created by combining a list of input-variables with an expression, like so:

```
let $max = #[$a $b] => if $a > $b then $a else $b
  in $max[25 12] // returns 25
```

In the example above, a lambda expression is assigned to to `$max`-variable, before being called as a function in line 2. The `#`-symbol is used to mark the beginning of a lambda expression. The scope rules of lambda expressions are further described in section 4.3 while the declaration of named functions is described in section 4.5.

Pattern

A unique feature of Junta is patterns. a pattern used for matching patterns on the board.

List

A list is an ordered collection of values. The same value may occur more than once.

Identifier

Action

Something about monads here...

4.3 Scoping

A scope is the context in which one or more variables exist. There are three types of scopes in Junta. Function scopes, lambda scopes and "let"-scopes.

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

Function scope

Consider a function definition such as:

```
define max[$a $b] if $a > $b then $a else $b
```

The variables `$a` and `$b` only exist within the function `max`. When calling the function:

```
max[5 23]
```

A new scope will be created and the values 5 and 23 are assigned to `$a` and `$b`, respectively.

Named functions (such as `max`) always exist in the global scope.

Lambda expression scope

When a lambda expression is created, a reference to the scope it was created in is saved with it. This is known as a closure, and means that a lambda expression may access variables outside of its own scope. The accessible variables are the variables that were available at the time of the creation of the lambda expression.

Consider the following example:

```
define getAdder[$a] #[$b] => $a + $b
```

A function `getAdder`, which takes one argument (`$a`) and returns a lambda expression, is defined. Notice how `$a` is used within the lambda expression. This means that when the lambda expression is created, it must remember the value of the variables that exist in the scope, in which it is created. The use of the `getAdder`-function could look like this:

```
let $adder = getAdder[25]
  in $adder[5] // returns 30
```

In the first line `getAdder` is called with the argument, 25. A new scope, A, is created, in which the variable `$a` is assigned the value 25. Then the function expression is evaluated, which results in a new lambda expression (with a reference to scope A). This is returned and assigned to `$adder` in line 1 of the above example.

In the second line the `$adder` is called as a function, which means that a new scope, B, is created, in which the variable `$b` is assigned the value 5. The important part is that B's parent scope is set to A (which is saved with the lambda expression). The expression (the right side of the lambda expression) is then evaluated. First the `$a`-variable is encountered. The interpreter first searches the B-scope for `$a`, and when unsuccessful, searches the parent-scope, A, for `$a`. In A the variable `$a` holds the value 25, and this is returned. Then the B-scope is searched for the `$b`-variable, and the value 5 is returned. The two integers are added, and the final return-value of the lambda-expression ends up beinng 30.

Let-expressions

JUNTA only supports *single assignment*. Single assignment is not assignment in the traditional imperative sense, but rather a way of binding a value to a symbol in a certain scope. This is done using *let-expressions*. Using a let-expression creates a new scope in which the declared variables are accessible. When leaving the scope the variables are destroyed.

The basic format of a let-expression is:

```
let VARIABLE1 = EXPRESSION1 in EXPRESSION2
```

In the example, the value of `EXPRESSION1` is assigned to `VARIABLE1`, which is available in `EXPRESSION2`. Another example could be:

```
let VARIABLE1 = EXPRESSION1, VARIABLE2 = EXPRESSION2 in EXPRESSION3
```

In this example, the value of `EXPRESSION1` is assigned to `VARIABLE2`, and the value of `EXPRESSION2` is assigned to `VARIABLE2`. Both `VARIABLE1` and `VARIABLE2` are available in `EXPRESSION3` and only in `EXPRESSION3`.

Destructive assignments are not possible JUNTA, meaning that it isn't possible to reassign a variable. It is however possible to hide a variable. Consider the following expression:

```
let $x = 5
  in (let $x = 6
    in $x + 2) + $x
```

The value of this expression is 13. This is because within the `$x + 2`-expression the `$x`-variable evaluates to 6. But in the outer expression `$x` evaluates to 5.

Nested *let*-scopes are possible. Consider for instance:

```
let $x = 4
  in let $y = 2
    in $x + $y
```

In the inner scope, both `$x` and `$y` are available. This is of course equivalent to:

```
let $x = 4, $y = 2
  in $x + $y
```

4.4 Operators

JUNTA supports a number of built-in operators. In this section the operators of JUNTA are described using the format:

`LeftOperandType operator RightOperandType → ReturnType`

The available types are described in section 4.2. Operations that are not described in this section can be considered invalid.

Boolean operators

These operators only accept boolean operands and only return boolean values.

- `Boolean and Boolean → Boolean`
  Returns true when both operands are true and false otherwise.
- `Boolean or Boolean → Boolean`
  Returns true when at least one of the operands are true and false otherwise.
- `not Boolean → Boolean`
  Returns true if the single operand is false and false otherwise.

Comparison operators

These operators are used when comparing two values, they will always return boolean values.

- `Integer < Integer → Boolean`
  Returns true if the left operand is less than the right one.
- `Integer > Integer → Boolean`
  Returns true if the left operand is greater than the right one.
- `Integer <= Integer → Boolean`
  Returns true if the left operand is less than or equal to the right one.
- `Integer >= Integer → Boolean`
  Returns true if the left operand is greater than or equal to the right one.
- `Integer == Integer → Boolean`
  `String == String → Boolean`
  `Boolean == Boolean → Boolean`
  `Coordinate == Coordinate → Boolean`
  `Direction == Direction → Boolean`
  Returns true if the left operand is equal to the right one.
- `Integer != Integer → Boolean`
  `String != String → Boolean`
  `Boolean != Boolean → Boolean`
  `Coordinate != Coordinate → Boolean`
  `Direction != Direction → Boolean`
  Returns true if the left operand is not equal to the right one.

Integer operators

The following operations are possible on integers:

- `Integer + Integer → Integer`
  Integer addition.
- `Integer - Integer → Integer`
  Integer subtraction.
- `Integer * Integer → Integer`
  Integer multiplication.
- `Integer / Integer → Integer`
  Integer division.

String operators

It is possible to concatenate strings:

- `String + String → String`
  Returns the concatenation of two strings.
- `String + Integer → String`
  `String + Boolean → String`
  `String + Coordinate → String`
  `String + Direction → String`
  `Integer + String → String`
  `Boolean + String → String`
  `Coordinate + String → String`
  `Direction + String → String`
  Returns the concatenation of a string and the string-representation of another type

Direction and coordinate operators

It is possible to combine directions using the following operator:

- `Direction + Direction → Direction`
  Combine two directions.
- `Coordinate - Coordinate → Direction`
  Returns the distance between two coordinates as a direction.
- `Coordinate + Direction → Coordinate`
  Add a direction to a coordinate.

For instance adding the directions `n` and `e` produces a direction equivalent with the direction `ne`. More information about the direction-type is available in **TODO:** section ????.

4.5 Functions

Functions are a big part of Junta, and are required for most calculations and operations.

In Junta functions are first-class citizens, meaning that functions are treated as any other value in the language. Therefore one can pass functions as arguments to other functions or returning them as values. It is also possible to create anonymous functions (using lambda expressions), and pass these to functions.

Standard environment

Many functions are made available to the programmer in Junta.

User-defined functions

A user can define custom functions for use in the implementation of a board game. This is done using the *define*-keyword. Functions can be declared to accept any number of parameters or none at all. An example of a function definition could be:

```
define max[$a $b] if $a > $b then $a else $b
```

It can be used as a way of putting frequently used expressions in one place.

4.6 Patterns

This section should cover how to use patterns and what to use them for. For instance there should be a list of operators and a comparison to regular expressions.

4.7 Abstract syntax

The abstract syntax is the interpreter or compiler's internal representation of a program. It is represented as an abstract syntax tree.

This section should cover all aspects of our abstract syntax tree, and how it differs from the parse tree.

Chapter 5

Implementation

5.1 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[11, 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[5, 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. [5, p.94].

Our scanner takes a raw source code for a program written in JUNTA as input. It validates the lexically correctness of a JUNTA program. The scanner tries to find tokens existing in JUNTA 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 JUNTA. **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 JUNTA 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 JUNTA:

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

The result of analysing the input can be seen in table 5.1. Tokens are needed for abstraction. When the parser later on will determine if the code respects the grammar of JUNTA, it is useful to have these abstractions. It makes it 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 JUNTA 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.

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

Table 5.1: *Analysing an input stream for lexemes and tokens.*

Our scanner contains 2 classes, **Scanner** and **Token**. **Token** contains an enum named **Type** that enumerates all the types of tokens in JUNTA. 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 5.1, 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 JUNTA. 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?

Listing 5.1: "The scan() method from the JUNTA scanner."

```
1 public Token scan() throws Exception {
2   while (isWhitespace()) {
```

```
3       pop();
4     }
5     if (isEof()) {
6       return token(Type.EOF);
7     }
8     if (isDigit()) {
9       return scanNumeric();
10    }
11    if (isUppercase()) {
12      return scanUppercase();
13    }
14    if (isOperator()) {
15      return scanOperator();
16    }
17    if (isLowercase()) {
18      return scanKeyword();
19    }
20    if (current() == '"') {
21      return scanString();
22    }
23    if (current() == '$') {
24      return scanVar();
25    }
26    throw new ScannerError("Unidentified character: " + current(), token(Type.EOF));
27  }
```

5.2 PARSER

In this section we present our handwritten parser for our programming language. We have written a recursive descent parser, which is within the class of LL(1) parsers. The grammar for our programming language is suiting for this because e.g. is does not have left-recursive productions.

The parser was very simple to implement, because it is structured in the same manner as the grammar is structured. For instance if the grammar expresses that the next set of terminals must begin with a left bracket ('[') then the parser will expect the next token to be a **LBRACKET**-token which is the token name for a left bracket.

In example 5.2 we give an example of how this structure looks like in the parser. The following production rule from our grammar shows what the if expression expects:

$$if\_expr \quad \rightarrow \quad \texttt{"if"}expression\texttt{"then"}expression\texttt{"else"}expression$$

The production for if expression says that every expression of this type must start with the combination of the two symbols `"i"` and `"f"` that spell the word `"if"`. When the parser meets this word it knows that it has to parse an if expression and the code this is reflected in example 5.2.

Listing 5.2: "This shows how if expressions are parsed."

```
1  private AstNode ifExpression() throws SyntaxError {
2    AstNode node = astNode(Type.IF_EXPR, "");
3    expect(Token.Type.IF);
4    node.addChild(expression());
5    expect(Token.Type.THEN);
6    node.addChild(expression());
7    expect(Token.Type.ELSE);
8    node.addChild(expression());
9
10   return node;
11 }
```

In example 5.2 the parser initialises the node for the expected if expression. The parser starts by calling the method **astNode** to create a node for the Abstract Syntax Tree (AST). We call the method with information about what

type of expression this is (**IF_EXPR**-token). The method calls the **expect**-method to verify that the next token is what we are expecting. If the two tokens do not match the parser throws a syntax error with information about the error.

Every grammar has a finite set of nonterminals and terminals that constitute the productions of the grammar. We have defined tokens for every nonterminal in the grammar. The if expression have the token name of IF_EXPR.

In the production for the if expression we have three terminals; the `"if"`, `"then"`, and the `"else"`. These are all expected in the method for any if expression. When the parser finishes reading a terminal it knows that the following token will be an expression, and therefore a new child for the node is made with a call to the **expression**-method wherein we parse expressions. Finally the method returns the node containing every child for the whole if expression.

We mentioned earlier that the parser is an LL(1) parser which means that the parser is able to look ahead in the sequence of tokens. We have shown the look ahead method to determine if the next token is part of an element. Recall that the productions for the nonterminal is as follows:

$$
\begin{array}{rcl}
element & \rightarrow & \texttt{"("}\ expression\ \texttt{")"} \\
& | & variable \\
& | & list \\
& | & pattern \\
& | & keyword \\
& | & direction \\
& | & coordinate \\
& | & integer \\
& | & string \\
& | & identifier \\
& | & function
\end{array}
$$

This means that an element can be quite a few things. This is why we have constructed a method to determine whether the next token is part of an element. This method is shown in example 5.3.

Listing 5.3: "This shows what the lookAhead method expects for an element as input."

```
1  private boolean lookAheadElement() {
2    return lookAheadLiteral()
3      || lookAhead(Token.Type.LPAREN)
4      || lookAhead(Token.Type.VAR)
5      || lookAhead(Token.Type.LBRACKET)
6      || lookAhead(Token.Type.PATTERNOP)
7      || lookAhead(Token.Type.KEYWORD)
8      || lookAhead(Token.Type.THIS)
9      || lookAhead(Token.Type.ID);
10 }
```

The method **lookAheadElement** makes use of two methods to figure out of the next token is part of an element. The first method is the **lookAhead**-method that takes a token as an argument and figures out if the next token in the sequence of tokens are equal to each other. The second method is like the method in example 5.3 but instead of checking for elements it checks for literals. Literals are one of four tokens; the **direction, coordinate, integer**-token and **string**-token. These are what the method **lookAheadLiteral** checks for. These methods return true or false.

In example 5.4 we show an example of how the **lookAhead**-method is used in the parser. The example is taken from the **expression**-method. Recall that the production for an expression is as follows:

$$
\begin{array}{rcl}
expression & \rightarrow & function\_call \\
& | & element\ operator\ expression \\
& | & if\_expr \\
& | & lambda\_expr \\
& | & element
\end{array}
$$

The grammar shows that an element can be followed by an operator or nothing. If the element is followed by an operator, then the operator is followed by a new expression. This must be reflected in the code of the parser.

Listing 5.4: "Use of the **lookAhead**-method. This example is from the **expression**-method."

```
1  else if (lookAheadElement()) {
2    AstNode element = element();
3    if (accept(Token.Type.OPERATOR)) {
4      AstNode operation = astNode(Type.OPERATOR, currentToken.value);
5      operation.addChild(element);
6      operation.addChild(expression());
7      node.addChild(operation);
8    }
9    else {
10     node.addChild(element);
11   }
12 }
```

The code in example 5.4 shows that the if-statement uses the **lookAheadElement**-method to determine if the next token is an element. If this is true, it enters the block of code inside the statement and creates a new node called "element" for this token. As we are inside the production for an expression the next thing the code checks for is if the next token after the **element**-token is the type of an operator. The **accept**-method returns true if the types correspond, and false if they don't. If the method returns true, the parser enters the block of code and adds a new node called "operation" with two children - the element-node just created and a new node called "expression", because the parser knows that the operator must be followed by an expression. Finally it adds the operation-node to the original node of the whole method called "node". However, if this is not the case, the code just return the new node called "element" as a child for the node of the node of the whole expression.

When the parser has parsed every token of the input it can produce an Abstract Syntax Tree that corresponds to the program written in the programming language. This shows that the parser is built very systematically according to the grammar of the programming language.

CHAPTER 6

EVALUATION

CHAPTER 7

CONCLUSION

Bibliography

[1] Stig Andersen. Den gode kravspecifikation. May 2006.

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

[3] c2. Domain specific language. http://c2.com/cgi/wiki?DomainSpecificLanguage. (22-03-2013).

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

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

[6] Will Freeman. Why board games are making a comeback. http://www.guardian.co.uk/lifeandstyle/2012/dec/09/board-games-comeback-freeman. (22-03-2013).

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

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

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

[10] Schoalistic. The benefits of board games. http://www.scholastic.com/parents/resources/article/creativity-play/benefits-board-games. (22-03-13).

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

[12] Stackoverflow. What is the difference between ll and lr parsing? http://stackoverflow.com/questions/5975741/what-is-the-difference-between-ll-and-lr-parsing. (2013-03-13).

[13] Étienne Gagnon. SableCC, an object-oriented compiler framework. http://sablecc.sourceforge.net/thesis/thesis.html, march 1998.

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

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

[16] Wikipedia. Kalah. http://en.wikipedia.org/wiki/Kalah. (2013-02-15).

[17] Wikipedia. List of programming languages. http://en.wikipedia.org/wiki/List_of_programming_languages. (22-03-2013).

[18] Wikipedia. JavaCC. http://en.wikipedia.org/wiki/JavaCC, March 2013.

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

Appendix

APPENDIX