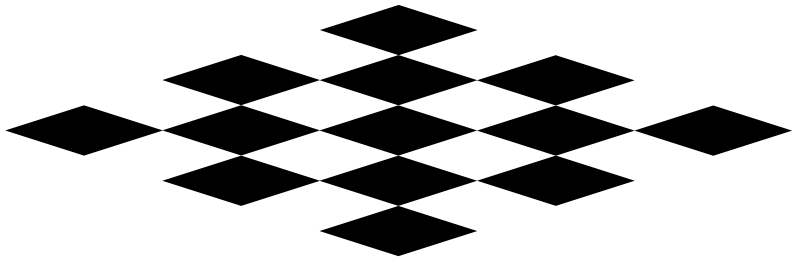
Aalborg University

Computer Science

Junta

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 – 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:

This report presents the process of defining, designing, and implementing the programming language called JUNTA. JUNTA is a purely functional object-oriented domain-specific programming language made specifically for programming board games.

In the process we have defined, designed, and implemented a scanner, parser, scope checker, and an interpreter. Furthermore, we have implemented a simulator, which through a game abstraction layer communicates with the interpreter and in which it is possible to play JUNTA games.

All of JUNTA's components are built on top of Java. This means that every system with Java virtual machine support also supports JUNTA.

Preface

This report describes the development of a domain-specific programming language called JUNTA which was designed to program playable board games. The specific topic was to design, define, and implement a programming language for generic game playing.

The project and the developed software, including all source code and this report, can be found at the following address:

```
https://bitbucket.org/Acolarh/p4-project/src/
```

Throughout this project we had been following three courses from which we have gained the needed knowledge and help to develop the programming language. The courses are: Languages and Compilers, Principles of Operating Systems and Concurrency, and Syntax and Semantics.

For this project we have developed a scanner, parser, scope checker and an interpreter, along with a graphical interface that makes playing games possible. These are all written in Java, and basic understanding for programming is needed to understand code examples. Furthermore, we also give examples of code written in JUNTA which is illustrated differently from the Java code. This will be explained once it is used in the report.

Reading guide

This report is divided into chapters wherein we have sections about specific subjects relevant to the project.

Chapter 3 presents a long list of requirements which JUNTA must respect. This chapter can be skipped without losing any important information to understand the following chapters. The requirements are derived from the previous chapter 2. The requirements are used as a checklist to make sure that we end up with a useful product.

Special thanks to

We wish to thank our supervisor for his patience and guidance throughout the project. We also wish to thank Hans Hüttel for reading semantics of JUNTA through and providing constructive criticism.

Contents

CHAPTER 1

A NEW PROGRAMMING LANGUAGE

At this very moment, hundreds maybe thousands of new programming languages are being created. A Wikipedia article about current programming languages that exist currently contains a list of more than 660 different programming languages[27], 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 so many other programming languages already exist, 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[2].

What we intend to do in this project is develop a programming language that can describe board games. That is designing, defining, and implementing it. We choose to develop a programming language for the domain in which the programming of board games lies, 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[18] and they can give a certain satisfaction[7]. 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 that makes it easy to program board games. Yes, it is possible to program board games in Java, C♯, and other general-purpose programming languages, but is it easy? Definitely 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. This can obviously done with game libraries, but these libraries adapt to the underlying language's data structures, which has implications of its own. 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 definitely 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? We must engage in some research about board games and the elements and components they consist of and we will need to analyse the tools and techniques necessary to implement a compiler. This is done in chapter 2.

Chapter 2

Analysis

This chapter focuses on the preliminary thoughts made before designing the language. Here research is shown and decisions on basic design principles are made. In section 2.1 we begin by defining what a board game actually is. What is the definition of it? Afterwards, in section 2.2 we analyse two board games, namely Chess and Kalah, to gain a better understanding of which elements and artefacts they contain. After the analysis of board games, we begin by analysing the four main programming paradigms in section 2.3 and how the different paradigms are used and what they are good for. After this we take a look at how syntactic analysis is done in section 2.4 and which methods we can use to analyse our own syntax. In this section, we for instance look at context-free grammars and different approaches to parsing and how a parser can be constructed. After this we look at which contextual constraints context-free grammars have in section 2.4.2. Furthermore, we take a look at code generation and interpretation of a given source code in section 2.6. Here we investigate the translation process. We also describe when an intermediate language can be used and what it is good for. In this section we also define what a compiler and an interpreter is. In section 2.7 we investigate and describe how a game simulator can be used to visualise the produced game in our programming language. Here we also describe which possible features we can add to make the simulator a good environment for the user.

This chapter must make it possible for us to make informed design decisions for our programming language designed in chapter 4. Lastly in section 2.8, we present our major decisions which have been made throughout this chapter.

2.1 What is a board game?

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 colour 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:

> "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."[29]

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 artefacts 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 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 games 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 easily in that language compared to how it would have been done in existing general-purpose programming languages.

To define what elements are to be included in our programming language (Junta), we believe it is essential to look at some existing board games. For this reason we have analysed two well-known board games. We have investigated the game elements that might cause trouble, or be clumsy, or not straightforward to implement in common general-purpose programming languages. In the analysis of the games a list of game elements is presented that respects all of the elements which we found interesting from the games. The aim of Junta is to ease the programming of these game elements.

2.2 Analysis of Chess and Kalah

In the following sections we will analyse the two games: Chess and Kalah. The reason for picking these two games in particular is because they are universially known games which we think contain some very fundamental board game elements that we need to know about to gain a better understanding of which features are needed in Junta.

2.2.1 Chess

Chess is a board game of two opposing players. Chess is a turn-based game 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. Kalah and Backgammon.

Each player has a total of 16 pieces: 8 pawns, 2 knights, 2 bishops, 2 rooks, 1 queen and 1 king. Each piece type has unique possible moves. 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 vertically or horizontally. The fact that each piece type has a unique way to move separates chess from a lot of other common board games where all pieces have the same abilities, like Noughts and Crosses, Kalah, Ludo, Backgammon. This means it should be possible to define a unique movement pattern for numerous different types of pieces in Junta.

Cut to the bone, Chess goes as follows: 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 take turn in which clever moves are being made and pieces are being 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 should be possible to look one move ahead to control if the checkmate condition is obtained.

Special moves

In chess, there are numerous special moves that don't follow the normal pattern of movements in chess. Earlier we mentioned that a pawn can move only one square vertically forward or one square diagonally when capturing an enemy piece. But this is not always the case. If the pawn is in its 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 following conditions must be met[4]:

1. The move has to be the very first move of the king and the rook
2. There can't be any pieces standing between the king and the rook
3. 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 ends up in

There exits two more special moves, called "En Passant" and "Promotion". These moves are important to the game, but not exactly relevant. Hence they are not going to be described here, but refer to [4] for more information.

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

So, what's 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 brief analysis above, we present this list of interesting game elements we found in chess:

- Pieces have 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"
- Constraints that disallow 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

From looking at chess, we conclude that these aspects and elements need to be easily done in JUNTA.

2.2.2 KALAH

Kalah is like chess, a turn-based game consisting of two opposing players. But 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 referred to as houses[26], with two of the houses separating themselves from the rest by being the houses or bases of each of the players. Furthermore, each player has six houses belonging to them (see figure 2.2). Each house (including the players' houses) can contain an arbitrary number of seeds, unlike in chess where the squares can only contain one piece.

Cut to the bone, Kalah goes as follows: When the game starts, each of the 12 houses contain 4 seeds (in some versions of the game each house contains 5 or 6 seeds), and the player bases are empty. Now the players take turn to pick up piles of seeds and deal them out to the 12 houses, including their 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), that is not empty and does not belong to the opponent, the player can pick up the pile of seeds in the house and start dealing these out. A player's turn ends if, when dealing out seeds, he lands in an empty house or one of the opponent's houses. For a more detailed description of the rules, we refer to [26]. The game is over once one of the players' six houses is empty and the winner player with most seeds in their base.

Figure 2.2: *The board game Kalah.*

Special moves

Like in Chess there are some special moves in Kalah that don'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 [26].

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

- Squares can contain an arbitrary number of pieces
- Making a move can be considered as simple as 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 non-empty square, the player can make another move
- A square can belong to a player
- Squares can be related to other squares
  - 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 squares belonging to a player are empty
- A winning condition - the player with the most pieces in their square wins when the end game condition has been met
- Only one type of piece

2.2.3 Chess and Kalah summary

We have now analysed the two board games Chess and Kalah and many interesting game elements have been recognised. For a programming language that allows these two games to be implemented, all of the game elements must be possible to implement in the programming language so this needs to be taken into consideration when designing Junta. Therefore the two list of interesting game elements in Chess and Kalah will be used to set up requirements for the design of our programming language. The list of requirements can be viewed in chapter 3.

2.3 Overview of the four main programming paradigms

In this section we will present the four main programming paradigms: imperative, object-oriented, declarative, and functional programming. It is not inconsequential to just develop a programming language before having a good understanding of the paradigms behind these languages, because we can learn a lot from them and use them to inspire Junta. We can simplify this argument by saying that each paradigm has its own area of expertise.

A programming paradigm describes a method and style of computer 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.

2.3.1 Overview

The four main paradigms are described as follows[11]:

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 functional programming languages is Haskell. Other examples of languages designed for functional programming are Erlang, F♯ and Lisp, while it is possible 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. We want to be able to have complete control over state changes in board games so following the functional paradigm and preventing side-effects will make it easier to manage state changes and the movement history of a game. Also, the object-oriented paradigm, has proved to be very useful for describing many real-word systems. Allowing the use of classes and objects may make it easier to create board games (e.g. an instance of class *Board* consists of many *Squares* containing *Pieces* belonging to *Players*). Combining these two paradigms we will get a functional object-oriented programming language for describing board games.

2.4 The syntactic translation phase

In this section we describe and analyse the phase where a compiler performs syntactic analysis. The syntactic analysis is performed by an algorithm called a parser. A parser checks whether or not the source code of a given program is set up syntactically correct according to the programming language it is written in. All programming languages have rules for how its tokens can be combined, described in section 2.4.1. A common way to describe these rules is by using formal language-generation mechanisms called grammars or context-free grammars, described insection 2.4.2).

There exists two main approaches to perform syntactic analysis. One is top-down parsing from which the family of LL(k) parsers derive (the k defines the number of tokens needed as look ahead). The other is bottom-up parsing from which the family of LR(k) parsers derive. In section 2.4.3, we analyse how LL- and LR-parsers work and we look at the advantages and disadvantages of each by comparing them. Furthermore, we look at different methods for making parsers, more specifically we analyse the pros and cons against writing the parser by hand vs. using parser-generating tools to generate a parser, presented in section 2.4.4.

2.4.1 Lexical analysis

Before the syntax analysis can be performed a scanner must perform lexical analysis. The scanner's job is basically to check a given source code for lexical errors and translating the input stream of characters from the source code into a stream of tokens which the parser can work with. This is done by identifying every lexeme in the source and attaching a potential token to it[6, p. 57].

Lexemes are strings of characters described by regular expressions. Typical examples of lexemes in a programming language are: variable names, integer literals, operators and special keywords etc. A variable name lexeme could be defined by the following regular expression: $[a-z, A-Z, \text{“\_”}][a-z, A-Z, 0-9, \text{“\_”}]^*$. Which means a variable name can start with either an upper case letter, lower case letter or an underscore followed by zero or more lower case letters, upper case letters, numbers or underscores. The syntax or semantics of regular expressions are not described here, we refer to Perl notation[16]. In table 2.1 we give examples of lexemes and the tokens they have been paired with. If both a and b are lexemes describing variable names and 102 and 42 are lexemes describing integer literals, then a and b or 102 and 42 can typically be used interchangeably and still give a syntactic meaningful program.

Lexemes	Tokens
x	VAR_NAME
random_var_name	VAR_NAME
RANdom_var_name2	VAR_NAME
1	INT_LITERAL
342	INT_LITERAL
52890	INT_LITERAL
+	PLUS_OPERATOR
-	MINUS_OPERATOR
*	MULT_OPERATOR
if	KEYWORD
while	KEYWORD
switch	KEYWORD

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

A scanner is a relatively simple component which can be constructed by writing it by hand or using a scanner-generating tool such as Lex, which generates an executable scanner by feeding it with a set of regular expressions. When implementing the scanner for JUNTA, we would likely benefit more from crafting the scanner by hand than by using a scanner-generating tool. By crafting it by hand we will know exactly how it is implemented. There might be some advantages of using a scanner-generating tool such as the fact that it can run faster due to optimized code, is more reliable, it is easier to maintain and it is faster to implement if one already knows how it works, if not, a handwritten scanner could be just as fast to write.

2.4.2 Context-free grammars

In the previous section, we described how the scanner transforms an input stream of characters into an output stream of tokens. In the following section we will describe context-free grammars, a component which contains the programming languages grammar, that is the set of rules for how its tokens can be combined.

A context-free grammar contains a set of terminals, a set of non-terminals, a set of productions and a start symbol. 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. A production typically has the following form:

$$non - terminal \quad \rightarrow \quad \{terminal \mid \ non - terminal\}$$

A start production specifies a single non-terminal, from where all syntactically valid strings, that are in the language, can be derived 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 successful, 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.

Contextual constraints for context-free grammars

Context-free grammars (CFGs), as explained in section 2.4.2, cannot describe syntax which can only be syntactically correct given a specific context (hence the name of these grammars). This means that there are constraints which a given CFG cannot describe. These contextual constraints are i.e.[12, p. 39]:

- Declaration before use
- Scope rules
- Type correspondence
- Overriding methods

The issue of remembering

The rule that variables must be declared before they can be used is impossible to express with CFGs. It would require that the CFG was able to remember things, particularly those variables that have been declared, which it cannot.

The problem of remembering things also shows up when working with scope rules. Typically, a variable declared in one scope cannot be used outside that scope, depending on scope rules, of course. The CFG cannot describe such scope rules that we describe as static semantic rules. It is named static because the analysis required to check the specifications can be done at compile time rather than run time[19, p. 153].

The same issue is present when working with type systems and the possibility of overriding methods in a programming language. The CFG must remember names of the variables, which type they have, and to be able to compare and devise a conclusion if a specific type is correct in the given context. Furthermore, the possibility of overriding methods is not possible because the CFG must remember method names, formal parameters, and return value to be able to formulate a production to make it possible to override the methods, which is not possible.

Solutions to this problem

The big issue for CFGs is to remember things. Are there any grammars that solve this issue? Yes there are. There exists different kinds of grammars which are able to describe the semantics of a given language instead of just focusing on the syntax. Grammars that can describe semantics are specified under a class of grammars called contextual grammars[12]. An example of such a grammar is an attribute grammar which actually "decorates" a CFG with those attributes we are interested in[22]. Contextual grammars can take the form of:

$$uAv \rightarrow uwv$$

Where u and v are in the context A is in at this transition. The issue with contextual grammars is that they are difficult to write, process, and there are no automated generators for efficient generation of parsers. These kinds of grammars are out of context and will not be taken much into account[22].

Summary for contextual constraints

The issue is that CFGs cannot remember what they have met in the past, and which context a given production must be in to be true. This makes is impossible to declare rules as declaration before use, scope rules, type rules, etc.

There exists contextual grammars that can describe a language given a specific context. The largest issue with these grammars is that it is not possible to automatically generate efficient translators.

2.4.3 LL- and LR-parsers

As mentioned in the introduction to the section, the LL-parsers derive from a top-down parsing approach. In terms of grammars, this means LL-parsers attempt to parse a string by starting at the start symbol of the grammar and through a series of left-most derivations match the input string. On the opposite, LR-parsers derive from the bottom-up parsing approach. Here LR-parsers attempt to parse by starting with the input string and through a series of reductions get back to the start symbol.

LL-parsers have 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 left-most predicted terminal. These two actions are continuously called until the entire input string has been eaten and thereby has been matched. An example of an LL(2)-parser can be seen in table 2.2. In the example the parser is based on the simple grammar:

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

	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: *An LL(2)-parser parsing the string "int + int".*

S, E and T are non-terminals, and $+$ and int are terminals.

LR-parsers also have two actions: shift and reduce. The shift action adds the next input symbol of the input string into a buffer for consideration. The reduce action reduces a collection of non-terminals and terminals into a non-terminal by reversing a production. These two actions are continuously called until the input string is reduced to the start symbol[19]. An example of an LR(1)-parser in action is illustrated in table 2.3.

	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: *An LR(1)-parser parsing the string "int + int".*

Comparison of the parsers

Compared to LL-parsers, LR-parsers are more complex and they are generally harder to construct[19, p. 193], but using automated generator tools this might not be the case. We take a look at how to construct a parser in section 2.4.4.

LR-parsers are more powerful than LL-parsers because they accept a larger variety of grammars. For instance LL-parsers can't handle grammars with left-recursion, while LR-parsers can. The "power" and complexity of a parser is very dependent on the number of look ahead tokens (k), which the parser makes use of. The larger k is, the more complex and difficult the parser is to construct, but the larger variety of grammars the parser also accepts. As illustrated in figure 2.3, LL-parsers are a proper subset of LR-parsers.

SLR stands for simple LR-parser, and LALR stands for Look-Ahead LR-parser.

2.4.4 Constructing a parser

The art of crafting parsers is very systematic and therefore different automated tools have been written to generate parsers for grammars that meet some specific standards. A grammar must for instance not be ambiguous, otherwise tools cannot make a distinct parser for the grammar. A grammar is ambiguous if more than one parse tree can be constructed the same string expressed in the language of that grammar.

In the following section, we analyse the elements of constructing a parser by hand and by use of a parser-generating tool. In the end we sum up the pros and cons of each of the methods.

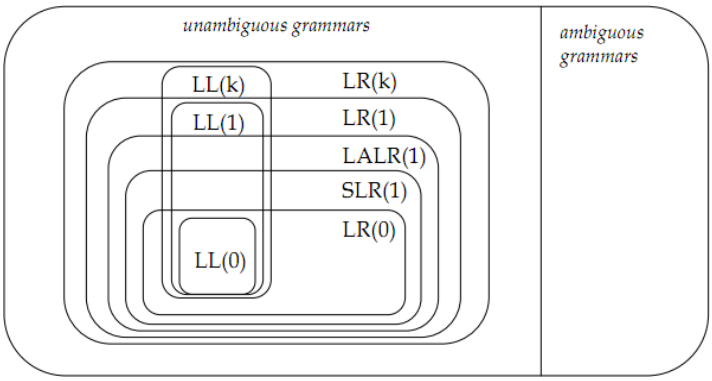


Figure 2.3: *Illustration of how the different parsers are connected. The figure is modified from slides presented in the "Languages and Compilers" course from Aalborg University in the spring of 2013.*

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 already built for this, it would provide us with a greater understanding of how parsers work. It is commonly known that one of the best ways to learn is by trial and error. There are some pitfalls in taking on this task. First of all, bugs are inevitable, when writing something new. These errors must be solved before the parser can be used. Therefore, the construction of the parser in this way, will be time-consuming.

Programming languages evolve and their grammars can change. When this is the case a parser must be maintained so that it still outputs the correct result. When we have constructed a parser by hand, this task will be time-consuming because we must search through the code of the parser and update it.

Whenever we work on correcting existing code we are 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. So far we have discussed that the produced code for a handwritten parser will be error-prone, which will naturally bring us to the conclusion that this code must be less reliable than the automated generators produced code. This is a big disadvantage because we have to be able to rely on the output of the parser, because if it fails, everything else that builds on top of it will too. To check the validity of a parser, it can be given a set of inputs that are almost in its language but contain a small error, which must cause the parser to reject the input. Additionally, a set of inputs can also be constructed, which is in the language, hence the parser must build a correct abstract syntax tree.

A big advantage for handwritten parsers is that we are 100% sure of what is written in the parser and we can customise the code for our specific needs. If we step back and look at this from a learning perspective, then this must be the optimal choice to take. Of course, this does not mean that there is nothing to be learned from using a tool. By using a tool we will learn to be critical against automated solutions and make decisions on whether or not a given tool would be optimal as a solution in the future.

Generated parsers

To construct a parser with a generator the developer must input a 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 expressed using the Extended Backus Naur Form (EBNF). A context-free grammar is on EBNF if it satisfies a certain set of rules and contains some special abilities. We will not describe these here, but for further information we refer to [6].

The generated parser will be produced in the language specified by the generator. An example of a parser generator is the SableCC parser project, which is a scanner and bottom-up parser generator, that generates LALR(1) parsers. This generator runs on the Java-platform and produces object-oriented code with clearly separated generated code. This contributes to the simplification and ease of maintaining the code[30, p. 11], clearly making it one of its selling points.

What differences are there between a handwritten and a generated parser? We need a grammar in both methods, so what are 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 (and maybe even a scanner) for us. Before we can use the software we have to figure out how to use it, but this is not a complicated process. Most software includes documentation on its features and usage.

The software that generates 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 most likely 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. Maintenance of the parser is also much easier because everything is machine-produced and can easily be changed to correspond with a grammar if it has been changed.

Though a disadvantage with generated parsers is that a lot of code will be generated without making much sense to the language designer, as we will see in section 5.2. To sit down and go through it all would be a daunting task. We could call this produced code for a black box because we really don't know what is inside it and how it works – it just does.

2.4.5 Summary of constructing scanners and parsers

From the above section about handwritten parser and parser generators we conclude the following advantages and disadvantages for handwriting a parser and generating a parser by a parser generator tool:

- Handwriting a parser means we gain a better understanding of how the parser work
- Handwriting a parser gives us the opportunity to customise the code specifically for our needs.
- Both handwritten parsers and generated parsers are time-consuming to construct
- Handwritten parsers are more time-consuming to maintain than generated parsers
- Handwritten parsers are less reliable than generated parsers
- Handwritten parsers are slower than generated parsers
- A generated parser is like a black box – we don't really know how it works

It is clear from the above list that there are many benefits in using a parser generator tool to construct a parser. It will be much easier to reach our goal by using such a tool and gaining experience in how they work can be a valuable skill. But we believe that it is also very important to gain experience in writing parsers ourselves, causing us to think about how they work from the beginning to the end. Furthermore, we can customise the code specifically for our needs. Therefore, we believe it will be beneficial to both write a parser by hand and generate a parser with a parser generator tool as well to see which one we want to build on top of.

2.5 Consideration of scope and type systems

In the following sections we take a quick look at the consequences of scope and type systems for Junta. We begin by introducing scope rules with a short discussion of the consequences of static and dynamic scoping. Afterwards, we introduce and discuss the consequences of static and dynamic type systems.

2.5.1 Which scope rules are we considering?

With static scoping the scopes of variables can be determined prior to execution. This means that a compiler can easily determine the type of every variable in the program by just examining the source code. So, when a variable is referenced in a statically scoped language, the value and type of the variable is the one it had at the time of declaration.

Static scope rules provide subprograms with access to non-local identifiers. This type of scoping works very well with compilers because the scope can be determined at compile time, which allows a compiler to do certain optimisations.

A disadvantage of static scoping is that it can give too much access and might need restrictions. But programs are dynamic and are often restructured which can lead to destruction of initial restrictions.

With dynamic scoping the scopes of variables can only be determined at run time, because it is based on the calling sequence of subprograms. When a variable is referenced in a language with dynamic scoping then the value and type of the variable is what it had on the time of the call to it[19, p. 227].

It is not possible to determine scopes statically, because the calling sequence of subprograms is not always known. When a method **A** calls a method **B**, then **B** has access to variables in that were declared in **A**. As a result dynamically scoped languages are much more difficult to read and understand, and this results in them being less reliable.

Comparing two similar programs with different scoping, then the statically scoped program will be much easier to read, more reliable, and it will execute much faster than the program written with dynamic scoping[19, p. 229].

2.5.2 Which type systems are we considering?

The type system is an essential part of the touch and feel of a programming language. There are two main type systems: the dynamic and the static type system. Whether to choose one over the other is today a very actively discussed topic. In the following section we list some advantages and disadvantages for each.

When it comes to detecting errors, static type systems make it possible to detect these in an early stage compared to the dynamic type systems. This is due to the fact that it is possible to check type errors already during compile time, rather than at run time. The readability is also improved by the static type system because of the presence of type names. This for instance makes it easier for the programmer to get an idea of what a certain subprogram is meant to do.

But the forced presence of types is also what decreases the writability of static type systems, since the programmer has to write down the types at all times, and when declaring variables, spend time considering whether or not a variable should be an integer type, a floating point or some other type, for example. This can take a lot of time for the programmer, instead of just letting the language calculate what is best suited when it compiles and runs the program.

So the dynamic type system is faster to write and it is more flexible, but the static type system can be easier to read and is more reliable at run time, since type checking is done at compile time. There is a big list of other advantages and disadvantages of each type system, which are explained in [13]. Here it is argued that perhaps a middle solution between the two type systems could be the optimal solution for a type system.

Which type system should we then go for in Junta? We are aiming to create a programming language in which it is easy and fast to create board games. It should be possible to create a board game with as few as possibles lines of code, and for that purpose the dynamic type system is suitable.

2.6 The interpretation and code generation phases

In this section, we present a brief overview of the phases of translators. Along with the design of our programming language Junta, we also want to make it possible for programmers who wish to write games in Junta to actually have the computer understand what they write. There are a number of ways we can make that happen. These are presented in the following subsections.

We begin by presenting the translation process followed by a presentation of what intermediate languages (IL) are and what they can be used for. In the subsequent section we present interpretation. Then we present a hybrid solution based upon compilation and interpretation. Lastly, we present what it will mean to have the game and engine separate, followed by a short summary, where we take some decisions based on the what we decide is important for our language.

2.6.1 Compilation

With compilation, an executable file is created for a specific platform that 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 connection would most likely be written. This engine would then be included directly in the executable upon compilation.

The translation process is typically not a simple task, therefore it is often split into different phases, shown in figure 2.4. The process can be split into many or few stages, depending on how detailed a process is desired and if an optimisation is done. In figure 2.4, the lexical and syntactical analysers make a lookup in a symbol table. Then the semantic analyser and the code generator use the symbol table to generate the correct code. Optimisation is optional in the phase of semantic analysis[19, p. 46].

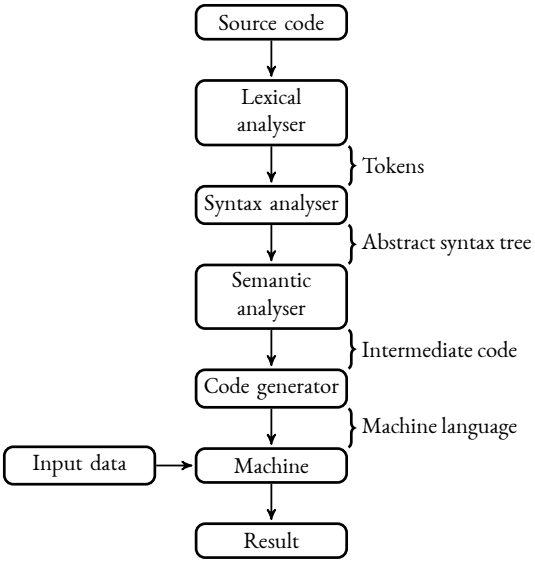


Figure 2.4: *The different phases of a compiler. Based on Sebesta et al.[19, p. 46, figure 1.3]*

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.

An intermediate language

A middle step between compiling or interpretation and generating executable machine code is to translate source code to an intermediate language (IL), which is then interpreted or compiled further. IL are usually more low-level than the initial source code, which would make it possible to optimize code for higher efficiency in later stages, such as eliminating superfluous node types and dead code.

Furthermore, one can compile to an intermediate language such as Java bytecode, executable on every machine that supports Java by having a Java virtual machine installed. This is very useful because the programmer, whom is developing a compiler for a language, does not have to construct a compiler for every platform; just one that translates to Java bytecode, that can then be translated further to many supported platforms. This way the programmer must only construct a single compiler. If one does not compile to an intermediate language, then the programmer must construct a compiler for each specific platform, which will be a lengthy and cost-heavy process. If you have m languages and n platforms, then the programmer must construct $m * n$ compilers to be able to compile to every platform. The difference between compiling directly to a platform or to an intermediate language and then further translating is illustrated in figure 2.5.

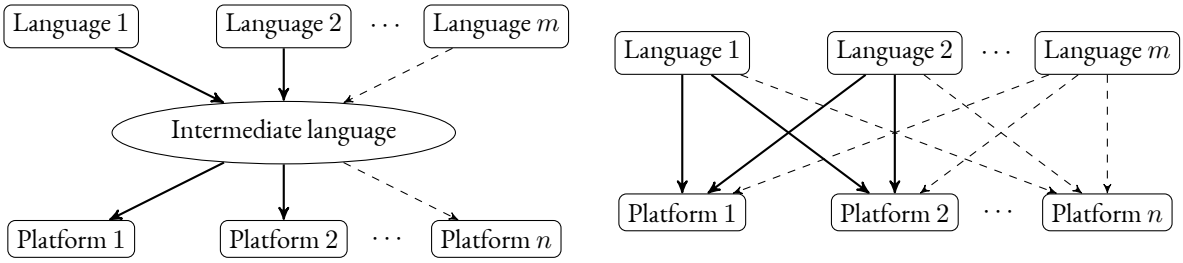


Figure 2.5: *Difference between compiling to an intermediate language.*

Now it is also possible to optimise the compiled source code before further translation. This way all code that has been compiled can be optimised, yielding better efficiency by noticing common patterns. An intermediate language

also gives the possibility to support multiple platforms and architectures, if you choose a popular and well supported IL. A well supported IL is a language that already has a compiler/interpreter built for the target platforms, saving the programmer from having to write them. Examples of such ILs can be high-level languages such as *C*, or low-level languages such as Microsoft's Common Intermediate Language bytecode or Java bytecode, that abstract away from platform-specific instructions and registers that other languages, such as assembly language use.

While Java bytecode is interpreted and therefore slower, modern interpreters use sophisticated methods such as Just-in-Time compilation (JIT), which at run time compiles intermediate code into native machine code. This process of course adds an overhead, and the speed differences are not that great anymore[10].

An intermediate format

To take it a step further, it could also be possible to use an intermediate format, which could be stored as an archive file that contains not only the code, but also sounds, images and other resources required to play a game. This for instance could make it easier to distribute games 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 wanted to share.

Using an non-existing intermediate format however means that you need to create a compiler, an interpreter, and the IL, which in turn requires a significant larger amount of work. Then this IL would need to further be compiled or interpreted so the machine understands it.

2.6.2 Interpretation

An alternative option is to write an interpreter. Interpreters take the original source code and execute each instruction at each translation. This means that a program will be parsed and executed on-the-fly when using an interpreter. It is required that the end-user has the interpreter. 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 2.6.4. The execution speed will however suffer and while techniques such as JIT exists to improve this, it is beyond the scope of this project.

A pure interpretation of a program lies at the opposite end from compilation in regards to methods of implementation. With this approach, which is illustrated in figure 2.6, no translation is performed at all.

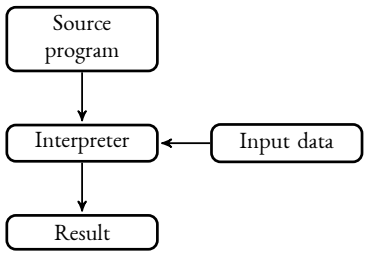


Figure 2.6: *The different phases of an interpreter. Based on Sebesta et al.[19, p. 48, figure 1.4]*

An interpreter literally "interprets" a program written in the targeted language. It acts like a virtual machine where instructions are statements of a 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 the location of faulty source code that caused the error. 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. Furthermore, the execution speed of an interpreter is usually 10 to 100 times slower than that of a compiler[19, p. 48].

2.6.3 Hybrid compilation and interpretation

The compiling or interpreting approach can be combined to form a hybrid implementation system. This method is illustrated in figure 2.7, 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, saving time for the programmer, since the error

will most likely ruin later stages anyway. 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[19, p. 50].

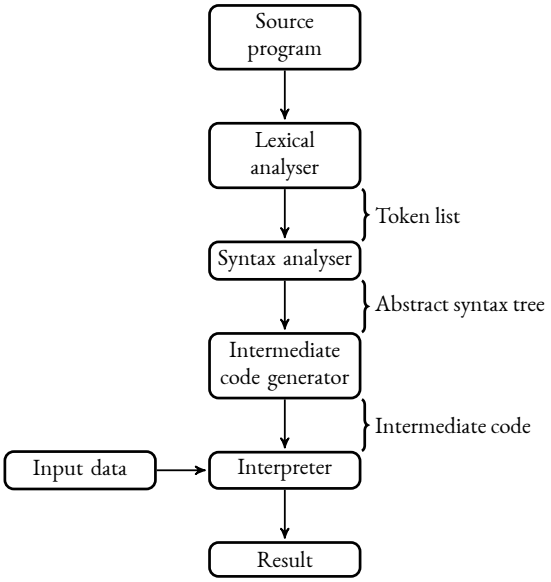


Figure 2.7: *The different phases of a hybrid implementation systems. Based on Sebesta et al.[19, p. 49, figure 1.5]*

2.6.4 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 adds 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 an updated version might never come out.

The disadvantage 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.

2.6.5 Summary of code generation and interpretation

The advantage of compilation is that the outputted code will run faster because a complete list of instructions will be ready to be executed. Although, a disadvantage is the time it takes to compile the code will take longer because the complete source code must be translated, though only once before executing the program can be done any number of times.

The advantage of interpretation is that it is possible to begin executing the program quickly because each instruction is interpreted on-the-fly which makes it faster than compiling the complete code.

If code is translated to an IL and then further translated, a lot of work away when talking about generating compilers to machine code because these compilers are platform dependant.

It is possible to combine compilation and interpretation. The program is compiled to 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.

We chose to interpret, because speed of executing a game is not a factor, and we want to be able to support the programmer as much as possible, providing detailed error messages if any exist.

2.7 A GAME 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 and a short summary.

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 programming language's interactive shell *irb*[17], 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[25]. 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.

2.7.1 WHY IS IT IMPORTANT TO HAVE A SIMULATOR?

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 setups. 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.

2.7.2 WHICH FEATURES ARE POSSIBLE AND UNDER CONSIDERATION?

We have decided 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 on 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 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.

Artificial Intelligence 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.

2.7.3 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 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.

2.7.4 Summary of simulating a board game

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 already simply be done from the source code. It would help make testing and playing games as authentic as possible.

2.8 Summary of essential decisions

Based on this chapter we have made the following decisions which our continued work will focus on.

When talking about crafting a parser we have chosen to craft an LL(1)-parser by hand and also generate a LALR(1)-parser with SableCC. The reason for this decision is the fact that this project is meant as a learning experience and therefore knowing how the parsing phase works is important. By both crafting a parser by hand and by the use of a generator tool we will get to know how a well-known parser generator works as well as construct our own customised parser.

We also presented the four main paradigms of programming languages and we will focus on combining the functional and object-oriented programming paradigms. These paradigms make great sense for a programming language that focus on board game programming. The reason for this is specified in section 2.3.

We will focus on crafting an interpreter rather than a compiler. This makes sense especially because we have decided to make a game simulator. By having an interpreter rather than a compiler, the game programmer will be able to make changes in the code and see them visualised in the simulator right away without having to recompile the whole program.

The decision of including a simulator is based on the assumption that board game programmers wish to have their games visualised and quickly be able to play and test them.

This chapter has set the foundation for a list of requirements for our programming language, which can be seen in the following chapter.

Chapter 3

Requirements

This chapter presents the requirements for our programming language which we have reached through our analysis. The requirements have been structured using a method published by Stig Andersen[1]. The requirements are used throughout the development phases and requirements are added as the project moves along and new challenges arise.

The following list of requirements is mainly useful for us as designers to make sure that the solution meets some requirements which express what the programming language must/can/should contain. The list does not tell us we wish accomplish this goal.

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 the project group define the correct solution[1].

Every requirement has been given a number so that it is possible later in the report to reference each requirement. This is for instance useful when we must conclude on our work at the end of the project.

Functional requirements:

1. The programming language will be used to program board games
    a) It must be possible to implement Chess, including the special rules of Chess
2. It must be possible to define what pieces the game consists of
3. It must be possible to define which pieces a specific player controls
4. It must be possible to define the possible squares a piece can be moved to
5. It must be possible to represent list structures
    a) It must be possible to perform list unions
    b) It must be possible to perform list intersections
6. It must be possible to use the language's built-in functions to do the following:
    a) Determine if a square is empty
    b) Determine if a square is occupied, and by who
    c) Check a condition for all objects in a collection
    d) Find all squares that match a specific pattern
    e) Concatenate lists
    f) Perform a lambda expression on each element in a list

   g) Move a piece to a square
   h) Capture the old piece on a specific square while moving a new piece to the same square
   i) Return which players turn it is
   j) Check if the current move about to be made for a piece is the first move made by that piece
7. It must be possible to determine which legal moves a player has
8. It must be possible to define winning conditions
9. It must be possible to define draw conditions
10. It must be possible to perform integer arithmetic
    a) Addition, subtraction, multiplication, and division
    b) The programming language must have boolean operators
    c) The programming language must have comparison operators
11. It must be possible to perform string concatenation
12. No function nor expression may produce side effects
13. There must be an action type that handles game state changes
14. It must be possible to create lambda expressions
15. It must be possible to declare functions
16. It must be possible to reference functions
    a) Functions must be first-class citizens
    b) It must be possible to call functions
17. The created board games must be playable in a graphical simulator
18. The simulator must be able to remember move history
    a) It must be possible to undo/redo moves
    b) It must be possible to save the move history
        i. It must be possible to start a game from a saved move history
    c) It must be possible to play over a network

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

Performance limitations:

19. The programmer must be able to implement board games with relatively few lines of code
20. The programming language must not be an extension of another programming language
21. It must be possible to have an arbitrary number of players
22. The source code of a single board game must be written in one file
23. The programming language must be formally defined
24. The programming language will be interpreted (not compiled)

Project limitations:

25. The programming language must be functional and operable no later than 29th of May, 2013
26. The group has approximately 20 hours per week to work on the project
27. The project is limited by the group members' skill in the design and development of programming languages

Solution goals:

28. The programming language must make it easy and quick for programmers to develop board games
29. The board games must be playable on different operating systems

We have listed requirements and limitations for this project. The requirements do not specify that the programming language must be able to perform I/O actions. This means that it is not a must that it is possible to do this.

An interesting requirement is that no functions can produce side effects which means that there cannot be any global variables. A side effect can happen when a global variable is manipulated in a function while the same global variable might be used outside the manipulating function.

Another thing about functions is that they must be first-class citizens, which means that they can be passed as arguments to other functions or as a return value.

Discussion

These requirements detailed above allow us to carry on designing and implementing the programming language, attempting to fulfill as many of the functional requirements and solution goals as possible during the course of development, always keeping them in the back of our minds.

The requirements that don't entirely relate to the project's focus described in chapter 1, such as being able to play over a network (requirement 19c), will not be weighted as much as other requirements, such as being able to declare functions (requirement 15).

Chapter 4

Design

In this chapter we present the design of our programming language, Junta. We begin this chapter by giving an example of a game implemented in Junta. The abstract syntax of Junta will be presented in section 4.1, followed by the lexical structure of programs in section 4.2. We also present the different expressions and their grammar in section 4.3 followed by the different definitions and their grammar in section 4.4. Futhermore, we introduce patterns in section 4.5 which are very important in Junta. Finally, we present the predefined types and constants of Junta in section 4.6, where the standard and game environments are presented.

In this chapter a number of terms are used, when referring to different aspects of the programming language. Essentially the language consists of *constants*, *functions* and *types*.

A type is a structure, that may inherit from a *super type*/*parent type* and contains *members* in the form of *data fields* and *constants*. A constant may be a function in which case it is referred to as a *method*. Types and constants can be *abstract*. A type can be instantiated using its *constructor*, this creates an *instance* or *object* of that type. All values in Junta are instances of types.

Code example

Junta is a purely functional object-oriented programming language designed explicitly for creating board games. We have developed Junta by first brainstorming and writing a handful of (partial) game implementations using a "programming language" which felt the most natural to us. This means that we were actually using the programming language before we had even constructed it. We began by writing programs in the unfinished language to try to find out how it should be built and what would be the easiest to write and understand.

In the following we go through the code sample (4.1). The example feature an implementation of the Noughts and Crosses game and introduces some of the important concepts of Junta. These will be further described in the rest of this chapter and in chapter 5.

The very first thing that is visible in code sample (4.1) are the two lines of comments. Comments are made with two forward slashes. Comments are described further in section 4.2.3.

The next thing that happens is the declaration of a type: `type NacGame[]`. The types of Junta can be compared to classes as seen in other object-oriented programming languages. The square brackets in Junta are used to encapsulate parameters and members of lists. `type NacGame[]` is a subtype of the super type `Game` which is a built-in type in Junta. The `extends` keyword is similar to the extends keyword of Java, and means the `NacGame[]` type inherits members of the `Game` type. `Game`'s constructor takes the title of the game as input and it contains many useful constants and functions, described further in section 4.6.

```
// An implementation of the traditional
// Noughts and Crosses game
type NacGame[] extends Game["Noughts and Crosses"] {
  define players = [
    NacPlayer[Crosses, "Crosses"],
    NacPlayer[Noughts, "Noughts"]
  ]
  define initialBoard = GridBoard[3, 3]
}
type NacPlayer[$pieceType, $name] extends Player[$name] {
  define winCondition[$gameState] =
    $gameState.findSquares[
      /friend (n friend n) | (e friend e) |
        (nw friend nw) | (ne friend ne ) friend/].size != 0
  define tieCondition[$gameState] =
    $gameState.board.isFull
  define actions[$gameState] =
  addActions[$pieceType[this], $gameState.board.emptySquares]
}
type Crosses[$owner] extends Piece[$owner]
type Noughts[$owner] extends Piece[$owner]
```

(4.1)

Constants and functions can be thought of as subprograms similar to methods. They distinguish themselves from each other by the fact that constants cannot take any parameters whereas functions can. Functions and constants are further explained in section 4.4.2. One of the built-in constants is `players` which contains a list of players. In the code sample at line four we see how the constant is defined. When a game written in JUNTA is played, the turn is shifted between each of the players in the list provided by the `players` constant. This is true, unless the turn order is specifically modified by the constant `turnOrder`, which is another built-in constant in the `Game` type. This however is not necessary in a Noughts and Crosses game, since the default turn order is desired. For instance, if the first player must be able to make three turns before the second player can make one, that would be defined in the `turnOrder`.

Furthermore, `Game` contains the constant `initialBoard` which in this case is assigned a grid board (another built-in type that takes the height and width of the board as parameters) of 3×3 squares. In other programming languages the override keyword is used when implementing methods from a super class but in JUNTA the override functionality already exists in the `define` keyword. For instance in code sample (4.1) `players` and `initialBoard` are overridden in line four and eight, respectively.

The next thing that is important in the code sample are the three functions: `winCondition`, `tieCondition` and `actions`. As the name indicates, the function `winCondition` checks if the current player is in a winning state and returns a boolean value: true or false. `winCondition` takes a game object as input. In Noughts and Crosses the win condition is obtained if a player has three of his pieces in a row in either a vertical, horizontal, or diagonal line. In JUNTA this is specified by using what we call "patterns", described in section 4.5. Patterns begin and end with forward slashes. A pattern for `winCondition` can be seen through line 13 and 14.

The function, `tieCondition`, checks if a tie is obtained and returns a boolean value: true or false. The tie condition is achieved whenever the board is full. This is specified using the built-in function: `isFull`.

The last function, `actions`, also takes as input a game object and contains a list of actions. In this case the only possible action is the function, `addAction`, which makes it possible to add a piece to the board of type `this`, which is the current player's piece type (crosses or noughts), to an empty square on the board.

At first sight the code sample will look complicated. This is mostly due to the overwhelming use of built-in functionality. This is however implemented to make it easier and faster for programmers to write code in JUNTA, since they don't have to implement all the functionality themselves.

Type system

In section 2.5.2 we analysed the two main type system approaches. We chose the dynamic type system due to the fact that it increases the writability of our programming language. The game seen in code sample (4.1) can be created in approximately 20 lines of code.

The type system in JUNTA comprises a number of simple types, from which every other type can be created. The simple types are: integers, character strings, booleans, lists, directions (vectors), coordinates (points), patterns, functions and types. Unlike in other programming languages, both functions and types in JUNTA are first-class citizens, this means that they can be used like any other value. This adds even more flexibility to the language. For instance in the Noughts and Crosses example (see code sample (4.1)) both the types `Noughts` and `Crosses` are passed as values to the constructor of `NacPlayer`. The simple types, their operators, and their methods are further explained in the rest of this chapter.

User types can be created using the `type` keyword. They are very similar to classes in traditional object-oriented languages, in that a type has a constructor, attributes, constants, and methods. An important aspect of the type system of JUNTA however, is that all values are immutable. It doesn't matter if it's an integer, a list, or an instance of a custom user type, the value of the object can't be changed. They can however be cloned and modified using various techniques depending on the type in question. For instance, adding two integers using the + operator returns a new integer representing the sum of the two operands. For instances of user types a new modified instance is returned when using the `set` keyword (this is further described in section 4.3.6 and section 4.4.3). The reason for this functionality, is to prevent side-effects, since randomly changing objects, could have undesirable influence on other functions or types that depend on these objects.

The type system of JUNTA supports single inheritance. An inheriting type will inherit all the members of its super type(s) (if `C` extends `B`, which extends `A`, then `C` inherits all members from both `B` and `A`). Visibility in JUNTA is implicit, in that all constants/methods are public (they can be accessed from anywhere as long as an instance of the type is available), while all data fields are private (they can only be accessed/changed from within the specific type, not even from inheriting types). Getters and setters are necessary in order to access data fields from the outside or in inheriting types. More details on data, inheritance, members, and abstract members are available in section 4.3.6 and section 4.4.3.

Another feature of JUNTA is implicit casting, when dealing with simple types (such as integers and strings). If a user were to create a type `MyInteger` extending the built-in type `Integer` (the type of integers in JUNTA), then instances of `MyInteger` could be used in place of simple integers. This works by casting the instance to a simple integer value (simply by throwing away the extra information provided by the `MyInteger` type). Explicit casting is only really possible with simple types, since their type constructors accepts one parameter of the same type. For instance the constructor `Integer` accepts a parameter of type `Integer`, meaning that it also accepts types that extend `Integer`. This makes it possible to cast a value of type `MyInteger` to a raw `Integer` value, albeit the usefulness of this functionality is dubious. The constructors of all the simple types implement this functionality however, shown later in section 4.6.1.

Scope rules

A scope is the context in which one or more variables or constants exist. In JUNTA we for instance have different expressions with their own scopes where their variables live and die. By this we mean that when the scope of the expression ends, the variables within the scope cannot be accessed anymore. These expressions with scopes will be defined in section 4.3.

Furthermore, it is important to know that JUNTA uses static scoping. The different kinds of scope rules were described in section 2.5.

4.1 Abstract syntax

Before we can describe the behaviour of programs written in JUNTA and their lexical structure, we must first present the syntax of programs. At this point, we are only interested in a notion of abstract syntax because we do not need to concern ourselves with operator precedence and so forth.

Syntactic categories

The abstract syntax of programming languages is defined as follows[9, p. 27]:

- A collection of syntactic categories
- For each syntactic category we have a finite set of formation rules that define how the elements of the given category can be built and combined

Table 4.1 presents the syntactic categories of JUNTA.

$n \in$	**Integer**
$x \in$	**Variable**
$s \in$	**String**
$e \in$	**Expression**
$p \in$	**Pattern**
$i \in$	**List**
$g \in$	**VarList**
$y \in$	**Coordinate**
$z \in$	**Direction**
$C \in$	**ConstantNames**
$T \in$	**TypeNames**
$F \in$	**FunctionConstants**
$D_G \in$	**GlobalDef**
$D_M \in$	**MemberDef**

Table 4.1: *The syntactic categories of JUNTA.*

In table 4.1 we have letters that define one syntactic category. For instance we have $n \in$ **Integer**, which means that when we see a n in the formation rules this is actually an integer value. There are two types of lists: **VarList** and **List**. These differ in that a variable list is used as formal parameters for types and the list is the sequence of parameters for a super type.

Furthermore, we have defined two definitions: **GlobalDef** and **MemberDef**. These are the set of definitions in their respective scopes. E.g. in the global scope we can define different constants and types. In the member scope we can also define different constants, and also the abstract definitions and the notion of data definitions. This will become very apparent in section 4.4, where we present the grammar of program structures.

Definitions

In this section we present the definitions which will be used throughout the construction of semantics for JUNTA. In the following definitions we use the syntactic categories presented in table 4.1. For some of the definitions we define arbitrary members which will be used in the semantics of the constructs of the language.

Definition (Type environment) The set of type environments is the set of partial functions from type names to type values:

$$\textbf{EnvT} = \textbf{TypeNames} \rightharpoonup \textbf{TypeValues}$$

An arbitrary member is defined as $env_T \in$ **EnvT**.

Definition (Constant environment) The set of constant environments is the set of partial functions from constant names to expressions and variable lists:

$$\textbf{EnvC} = \textbf{ConstantNames} \rightharpoonup \textbf{Expression} \times \textbf{VarList}$$

An arbitrary member is defined as $env_C \in$ **EnvC**.

Definition (Variable environment) The set of variable environments is the set of partial functions from variables to values:

$$\textbf{EnvV} = \textbf{Variable} \rightharpoonup \textbf{Values}$$

An arbitrary member is defined as $env_V \in$ **EnvV**.

Definition (Values) The set of values can contain many different values, and is defined as follows:

$$\begin{aligned}\textbf{Values} = \textbf{Integers} \cup \textbf{Strings} \cup \textbf{Lists} \cup \textbf{Patterns} \cup \textbf{Coordinates} \cup \textbf{Directions} \\ \cup \textbf{TypeValues} \cup \textbf{FunctionValues} \cup \textbf{ObjectValues} \cup \textbf{Booleans}\end{aligned}$$

Definition (List values) The values of lists is defined as follows:

$$\textbf{ListValues} = \textbf{Integers} \times \textbf{Elements}$$

The length of a list is an arbitrary member of $l \in$ **Integer**.

Definition (List elements) The set of elements in lists is the set of partial functions from integers to values:

$$\textbf{Elements} = \textbf{Integers} \rightharpoonup \textbf{Values}$$

An arbitrary member is defined as $elem \in$ **Elements**.

Definition (Function values) The set of function values is defined as follows:

$$\textbf{FunctionValues} = \textbf{VarLists} \times \textbf{Expressions} \times \textbf{EnvV} \times \textbf{EnvC}$$

A function value consists of a variable list, an expression and the set of variable and constant environments.

Definition (Type values) The set of type values is defined as follows:

$$\textbf{TypeValues} = \textbf{TypeNames} \times \textbf{VarLists} \times \textbf{D}_\textbf{M} \times \textbf{List} \times \textbf{TypeValues}$$

An arbitrary member is defined as $t \in$ **TypeValues**.

A type value consists of a type name, a variable list, the member definitions, the super type's paramters and the super type's type value. A type value is recursively defined since **TypeValues** is defined in terms of itself. This is not a problematic definition because these must be considered as values for two different types. The **TypeValues** on the right-hand side is in fact the super type's set of values whereas the **TypeValues** on the left-hand side is the current type's set of values. The list in the definition is in fact the super type's parameters whereas the variable list is the current type's formal parameters.

Definition (Object values) The set of object values (an instantiated **TypeValue**) is defined as follows:

$$\textbf{ObjectValues} = \textbf{TypeValues} \times \textbf{EnvC} \times \textbf{EnvV} \times \textbf{ObjectValues}$$

An object value consists of the set of type values, the set of constant environment, the set of variable environments and the set of object values. This is yet another recursively defined definition, because an object value can have a parent object value, this is the case when an object value extends another object value.

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid s \mid y \mid z \mid T \mid C \mid i \mid \texttt{-}e \mid \texttt{(}\ e\ \texttt{)} \mid \texttt{/}\ p\ \texttt{/} \mid \\
 & & e\ i \mid e\texttt{.}C \mid \texttt{\#}\ g\ \texttt{=>}\ e \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{not}\ e \mid e_1\ F\ e_2 \mid \\
 & & \texttt{let}\ x_1\ \texttt{=}\ e_1,\ \ldots,\ x_k\ \texttt{=}\ e_k\ \texttt{in}\ e_{k+1} \mid \texttt{set}\ x_1\ \texttt{=}\ e_1,\ \ldots,\ x_k\ \texttt{=}\ e_k \mid \texttt{this} \mid \texttt{super} \\
F & ::= & \texttt{is} \mid \texttt{and} \mid \texttt{or} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \\
 & & \texttt{/} \mid \texttt{\%} \\
i & ::= & [e_1,\ \cdots, e_k] \\
g & ::= & [x_1,\ \cdots,\ x_k] \mid [x_1,\ \cdots,\ \ldots x_k] \mid [\ldots x] \\
D_G & ::= & \texttt{define}\ C\ \texttt{=}\ e\ D_G \mid \texttt{define}\ C\ g\ \texttt{=}\ e\ D_G \mid \texttt{type}\ T\ g\ D_G \mid \\
 & & \texttt{type}\ T\ g\ \texttt{extends}\ T\ i\ D_G \mid \texttt{type}\ T\ g\ \{D_M\}\ D_G \mid \\
 & & \texttt{type}\ T\ g\ \texttt{extends}\ T\ i\ \{D_M\}\ D_G \mid \varepsilon \\
D_M & ::= & \texttt{define}\ C\ \texttt{=}\ e\ D_M \mid \texttt{define}\ C\ g\ \texttt{=}\ e\ D_M \mid \texttt{define abstract}\ C\ D_M \mid \\
 & & \texttt{define abstract}\ C\ g\ D_M \mid \texttt{data}\ x\ \texttt{=}\ e\ D_M \mid \varepsilon
\end{array}
$$

Figure 4.1: *The formation rules for the syntactic categories of* Junta.

Formation rules

Each syntactic category is used in one or more of the formation rules presented in figure 4.1. The formation rules define the structure of the members of the syntactic categories.

Not all of the constituents of the formation rules are syntactic categories. We for instance see different parentheses, forward slashes, different operators, and words like `this` and `define`. These are part of the construction of the given formation rule. If they are omitted from a rule then it is not valid in Junta.

The ::= means that the left-hand side of the rule can be any one of the | -separated right-hand sides. Furthermore, we use "· · · " to illustrate a repetition of some element in the rule. We have also used the slightly different ". . . " to illustrate the three dots that precede a variable argument (vars) which we present in section 4.4.2. It should be clear from the context which of the two is being used.

Epsilon denoted by ε represents an empty definition.

4.2 Lexical structure

This section presents the low-level non-terminals of Junta. We begin by describing the conventions we use throughout the report. Then we describe the contents of the lexical structure of Junta, such as the different reserved keywords, identifiers, and literals that Junta contains.

4.2.1 Notational Conventions

We use a variant of Extended Backus-Naur Form (EBNF) 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"`.

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

The following operations are used throughout this section to describe the grammar of the programming language:

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

The left-hand side is what will be seen in the grammar, while the right-hand side is a description of what the operation means. For instance a terminal will always consist of `" "` around the name of the terminal.

4.2.2 Character Classes

To be able to describe which characters or symbols a non-terminal can consist of in a concise manner, we need to define some sets of symbols to specific names which we can use in the description of our grammar.

The following classes of characters will be used throughout this section:

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

For instance a string character ($strchar$) can by any unicode character besides a quotation mark or a backslash.

4.2.3 Comments

In Junta a single-line comment begins with a sequence of at least two forward slashes (`//`). The comment ends at the next newline. Everything after the first two forward slashes and until the first next newline is completely ignored beyond lexical analysis. Comments are valid white space.

The following example shows a valid comment within an expression:

```
$a - 23 // a valid comment
  + $b
```
(4.2)

Unlike other programming languages, Junta does not have support for multi-line comments (such as `/* */` in C-like languages).

4.2.4 Reserved keywords

The following list presents reserved keywords in Junta. These keywords cannot be used as identifiers in the language, which is apparent in the following section.

$$
\begin{array}{rcl}
reserved & \rightarrow & \texttt{"define"} \mid \texttt{"type"} \mid \texttt{"abstract"} \mid \texttt{"data"} \mid \texttt{"extends"} \mid \texttt{"let"} \mid \texttt{"in"} \\
& \mid & \texttt{"set"} \mid \texttt{"if"} \mid \texttt{"then"} \mid \texttt{"else"} \mid \texttt{"not"} \mid \texttt{"and"} \mid \texttt{"or"} \mid \texttt{"this"} \mid \texttt{"super"} \\
& \mid & \texttt{"foe"} \mid \texttt{"friend"} \mid \texttt{"empty"} \mid \texttt{"is"}
\end{array}
$$

4.2.5 Identifiers

Junta has three different identifiers which are defined as follows:

$$
\begin{array}{rcl}
constant & \rightarrow & (\ lowercase\ \{alphanum\}\ )\ -\ (\ reserved \mid direction\ ) \\
type & \rightarrow & (\ uppercase\ \{alphanum\}\ )\ -\ coordinate \\
variable & \rightarrow & \texttt{"\$"}\ alphanum\ \{alphanum\}
\end{array}
$$

Constants, types, and variables cannot be defined with equal names because they begin with different characters.

4.2.6 Literals

The following list presents the literals of Junta:

$$
\begin{array}{rcl}
integer & \rightarrow & decimal\ \{decimal\} \\
direction & \rightarrow & \texttt{"n"} \mid \texttt{"s"} \mid \texttt{"e"} \mid \texttt{"w"} \mid \texttt{"ne"} \mid \texttt{"nw"} \mid \texttt{"se"} \mid \texttt{"sw"} \\
coordinate & \rightarrow & uppercase\ \{uppercase\}\ \ decimal\ \{decimal\}
\end{array}
$$

The string literal can contain any unicode character if it is escaped. This lets us construct any string. For instance if one needs to use a quotation mark within a string, it is possible by escaping the quotation marks.

$$
\begin{array}{rcl}
string & \rightarrow & \texttt{'"'}\ \{strchar \mid escape\}\ \texttt{'"'} \\
escape & \rightarrow & \texttt{"\textbackslash"}\ unichar
\end{array}
$$

4.3 Expressions

Throughout this section we will present all the expressions that are included in Junta. We have provided big-step semantics for six central expressions in this section. These are the ones that we have deemed most necessary and they are not as any other construct. Which sections that include semantics will be explained in the following.

The smallest parts of expressions are presented in section 4.3.1. In section 4.3.2 the notion of lists is presented and explained. Here we present big-step semantics. Then we present the let expressions in section 4.3.3. Also here we present big-step semantics. Afterwards, we present the conditional expressions in section 4.3.4 followed by a lambda expressions in section 4.3.5. We provide big-step semantics for lambda expressions. Furthermore, we present the concept of a set expression in section 4.3.6. We also supply big-step semantics for set expressions. Lastly, we present the different operators and calls in section 4.3.7. Here we provide big-step semantics for function calls and for member access.

We begin here by listing the main expressions below this paragraph:

$$
\begin{array}{rcl}
expression & \rightarrow & let\_expr \\
& \mid & if\_expr \\
& \mid & set\_expr \\
& \mid & lambda\_expr \\
& \mid & \texttt{"not"}\ expression \\
& \mid & lo\_sequence
\end{array}
$$

Two statements hold about expressions in Junta:

1. An expression **always** has a value.
2. An expression **cannot** have side effects.

4.3.1 Atomic expressions

These are the smallest possible parts of expressions in Junta, defined by the rule:

$$
\begin{array}{rcl}
\textit{atomic} & \rightarrow & \texttt{"("}\ \textit{expression}\ \texttt{")"} \\
& | & \textit{constant} \\
& | & \textit{type} \\
& | & \textit{variable} \\
& | & \texttt{"this"} \\
& | & \texttt{"super"} \\
& | & \textit{integer} \\
& | & \textit{string} \\
& | & \textit{direction} \\
& | & \textit{coordinate} \\
& | & \texttt{"/"}\ \textit{pattern}\ \texttt{"/"} \\
& | & \textit{list}
\end{array}
$$

The first atomic expression is `"("` *expression* `")"`, which means that it is possible to embed expressions within other expressions, and manually control the precedence of operations. Consider the following two expressions:

```
3 * (23 + 2) // = 75
```
(4.3)

```
(3 * 23) + 2 // = 71
```
(4.4)

In code sample (4.4) the parentheses are in fact unnecessary because the `*` operator has precedence over the `+` operator (see section 4.3.7).

Names (constants, types, and variables) are also atomic expressions, and are evaluated to whatever value they are associated with, based on the current scope. The keywords (`"this"` and `"super"`) are atomic, but only applicable within type definitions, where `"this"` refers to the current object and `"super"` refers to the current object casted to its parent type (if it has one).

The literals (*integer*, *string*, *direction*, and *coordinate*) are evaluated to their respective values, while patterns are evaluated to `Pattern` values according to the grammar in section 4.5 and lists are evaluated to `List` values according to the grammar in section 4.3.2.

4.3.2 Lists

The `List` type is one of the basic types in Junta. A `List` value is created using the following syntax:

$$
\textit{list} \quad \rightarrow \quad \texttt{"["}\ [\ \textit{expression}\ \{\texttt{","}\ \textit{expression}\}\ ]\ \texttt{"]"}
$$

Essentially this means that a list is created from zero, one, or more expressions (separated by commas). The following statements can be made about lists:

1. A list can be empty: `[]`
2. Lists begin at offset 0
3. Lists are ordered (`[1, 2]` $\neq$ `[2, 1]`)
4. Lists are immutable

When a list is evaluated, each expression is evaluated to a value (the order of evaluation does not matter, since no side-effects are possible, lazy-evaluation of expressions could even be a possibility), and all values (in the same order as the expressions) are added to the resulting `List` value. When a `List` value is created, it can't be altered further (because of the no-side-effects condition). All operations on that `List` value will create new `List` values, and leave the original value intact.

Values within lists can be accessed using the `[]` operation (same as with function calls and type instantiation). Because lists begin at offset 0, in order to access the second element of a list, one write as illustrated in code sample (4.5).

```
[25, 5, 17, 3][1] // => 5
```
(4.5)

Ranges of elements can also be returned. For instance in code sample (4.6) a new list is returned containing elements from offset 1 up to and including offset 2 (the list `["is", "a"]`).

```
let $myList = ["this", "is", "a", "list"]
in $myList[1, 2] // =>["is", "a"]
```
(4.6)

An offset can be negative, which means that the offset is dependent on the size of the list. This way offset -1 will always refer to the last element of the list, because the length of the list minus 1 will give the offset of the last element. For example in order to return the last two elements of a list one could use the offsets, -2 and -1:

```
[5, 2, 3, 4, 7][-2, -1] // => [4, 7]
```
(4.7)

Some other examples are presented in code sample (4.8):

```
[3, 5, 6, 1, 2][1, -1] // => [5, 6, 1, 2]
[3, 5, 6, 1, 2][-1] // => 2
[3, 5, 6, 1, 2][-4, -3] // => [5, 6]
[3, 5, 6, 1, 2][1, 2] // => [5, 6]
```
(4.8)

Big-step semantics

The semantics presented in table 4.2 are the transition rules for lists.

List access requires two rules, since two cases are possible. The first case, is when just one offset is requested, then that element should be returned. In the second case, two offsets are requested, and a range of elements should be returned, also in the form of a list.

4.3.3 Let expressions

Variables in Junta are assigned using let expressions:

$$
\begin{aligned}
\mathit{expression} &\rightarrow \cdots \mid \mathit{let\_expr} \\
\mathit{let\_expr} &\rightarrow \texttt{"let"}\ \mathit{variable}\ \texttt{"="}\ \mathit{expression}\ \{\texttt{","}\ \mathit{variable}\ \texttt{"="}\ \mathit{expression}\} \\
&\quad\ \texttt{"in"}\ \mathit{expression}
\end{aligned}
$$

A let expression consists of one or more assignments and an expression. Each assignment assigns the value of an expression to a variable name. These variables can then be used in the expression after the `in` keyword. After the evaluation of the let expression, the variables cease to exist.

$$[\text{LIST}_{\text{ACCESS-1}}] \quad \frac{env_T, env_C, env_V \vdash \langle e\ \rangle \rightarrow v_2 \qquad env_T, env_C, env_V \vdash \langle i \rangle \rightarrow v_3}{env_T, env_C, env_V \vdash \langle e\ i \rangle \rightarrow v_1}$$

where $v_2 = (l_1, elem_1)$
and $v_3 = (l_2, elem_2)$
and $l_2 = 1$
and $d = elem_2\ 0$

$$\text{and } v_1 = \begin{cases} elem_1\ d & \text{if } d \geq 0 \\ elem_1\ (l_1 + d) & \text{if } d < 0 \end{cases}$$

$$[\text{LIST}_{\text{ACCESS-2}}] \quad \frac{env_T, env_C, env_V \vdash \langle e \rangle \rightarrow v_2 \qquad env_T, env_C, env_V \vdash \langle i \rangle \rightarrow v_3}{env_T, env_C, env_V \vdash \langle e\ i \rangle \rightarrow v_1}$$

where $v_2 = (l_1, elem_1)$
and $v_3 = (l_2, elem_2)$
and $l_2 = 2$

$$\text{and } d = \begin{cases} elem_2\ 0 & \text{if } elem_2\ 0 \geq 0 \\ l_1 + elem_2\ 0 & \text{if } elem_2\ 0 < 0 \end{cases}$$

$$\text{and } j = \begin{cases} elem_2\ 1 & \text{if } elem_2\ 1 \geq 0 \\ l_1 + elem_2\ 1 & \text{if } elem_2\ 1 < 0 \end{cases}$$

$$\text{and } elem_3\ z = \begin{cases} elem_1\ d+1 & \text{if } z = 0 \\ \vdots & \\ elem_1\ d+n-1 & \text{if } z = n-1 \end{cases}$$

and $n = j - d + 1,\ elem_3$
and $v_1 = n$

Table 4.2: *Transition rules for accessing lists.*

Informal scope rules for let expressions

Destructive assignments are not possible in Junta, meaning that it isn't possible to reassign a variable. It is however possible to hide a variable. Consider the expression presented in code sample (4.9).

```
let $x = 5
  in (let $x = 6
    in $x + 2) + $x
```
(4.9)

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 code sample (4.10).

```
let $x = 4
  in let $y = 2
    in $x + $y
```
(4.10)

In the inner scope, both `$x` and `$y` are available. This is of course equivalent to code sample (4.11).

```
let $x = 4, $y = 2
  in $x + $y
```
(4.11)

Big-step semantics

The semantics presented in table 4.3 are the transition rules for the let expression. This transition rule is defined recursively to best illustrate the functionality of the expression.

[LET-1] $$\frac{env_T, env_C, env_V \vdash e_1 \rightarrow v_1 \quad env_T, env_C, env_V[x_1 \mapsto v_1] \vdash \langle \texttt{let } x_2 = e_2, \cdots, x_k = e_k \texttt{ in } e_{k+1} \rangle \rightarrow v_{k+1}}{env_T, env_C, env_V \vdash \langle \texttt{let } x_1 = e_1, x_2 = e_2, \cdots, x_k = e_k \texttt{ in } e_{k+1} \rangle \rightarrow v_{k+1}}$$

where $k \geq 2$

[LET-2] $$\frac{env_T, env_C, env_V \vdash e_1 \rightarrow v_1 \quad env_T, env_C, env_V[x_1 \mapsto v_1] \vdash \langle e_2 \rangle \rightarrow v_2}{env_T, env_C, env_V \vdash \langle \texttt{let } x_1 = e_1 \texttt{ in } e_2 \rangle \rightarrow v_2}$$

Table 4.3: *Transition rules for let expressions.*

The transition rules for [LET-1] are recursive because we must evaluate each expression ($x_1 = E_1$) individually before we move on to the next one. This is a must because of the fact that the next expressions can in fact make use of the previous expressions value. As an example take a look at the following code sample:

```
let $a = 2, $b = $a * 2
  in $b + $a
```
(4.12)

So, each call where there is more than one expression to be evaluated, the transition rule [LET-1] where $k \geq 2$ is used. Here the expression first in line to be evaluated will be evaluated before a new call to one of the two transition rules is made. When we reach a let expression with only one expression, we call the transition rule [LET-2] where $k < 2$.

4.3.4 Conditional expressions

It is often desirable to base the result of an expression on some sort of condition. In Junta this is achieved bye using *if* expressions, as defined by the following syntax:

$$\begin{aligned} expression &\rightarrow \cdots \mid if\_expr \\ if\_expr &\rightarrow \texttt{"if"}\ expression\ \texttt{"then"}\ expression\ \texttt{"else"}\ expression \end{aligned}$$

Unlike in most imperative languages, the conditional construct in Junta is not a statement (Junta doesn't have statements) but an expression. Since all expressions must have a value, the *else* part of en if expression is compulsory.

The if expression first evaluates the condition (the first expression). The resulting value must be of type `Boolean`. If the value is equal to the boolean true value, the *then* expression is evaluated, and the result is returned. If the value is false, then the *else* expression is evaluated, and the result returned.

4.3.5 Lambda expressions

Lambda expressions are expressions that evaluate to anonymous functions. In Junta they are defined as:

$$\begin{aligned} expression &\rightarrow \cdots \mid lambda\_expr \\ lambda\_expr &\rightarrow \texttt{"\#"}\ varlist\ \texttt{"=>"}\ expression \end{aligned}$$

The non-terminal $varlist$ represents a list of formal parameters, and is further explained in section 4.4.2.

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 example in code sample (4.13).

```
define getAdder[$a] = #[$b] => $a + $b
```
(4.13)

The function `getAdder` takes one argument (`$a`) and returns a lambda expression. Notice how `$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 be as illustrated in code sample (4.14).

```
let $adder = getAdder[25]
in $adder[5] // returns 30
```
(4.14)

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 code sample (4.14).

In the second line, `$adder` is called as a function, meaning that a new scope, B, is created where the variable `$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 being 30.

BIG-STEP SEMANTICS

The semantics presented in table 4.4 is the transition rule for lambda expressions.

[LAMBDA]	$env_T, env_C, env_V \vdash \langle \texttt{\#}\ g\ \texttt{=>}\ e \rangle \rightarrow v$	where $v = (g, e, env_V, env_C)$

Table 4.4: *Transition rules for lambda expressions.*

The three environments (env_T, env_C, env_V) must be known before it is possible to execute a lambda expression. We need to know which types, constants, and different variables are given in the specific scope.

The lambda expressions evaluates to a value v. The side condition of the transition rule explains that v is assigned the 4-tuple, a function value.

4.3.6 SET EXPRESSIONS

Set expressions look a bit like let expressions, but are only applicable within type definitions:

$$
\begin{aligned}
\mathit{expression} &\rightarrow \cdots \mid \mathit{set\_expr} \\
\mathit{set\_expr} &\rightarrow \texttt{"set"}\ \mathit{variable}\ \texttt{"="}\ \mathit{expression}\ \{\texttt{","}\ \mathit{variable}\ \texttt{"="}\ \mathit{expression}\}
\end{aligned}
$$

Set expressions are used to "modify" the value of data members in objects (see section 4.4.3 for an explanation of data). Each variable in the set expression must exist in the current type as data members. Since modifying a data member would be a side-effect, which is not allowed, the set expression instead returns a clone of the object, with the specified data members set to their respective values. This is useful for making setters (or something that looks like setters). Consider for example presented in code sample (4.15).

```
type MyType[] {
  data $myData = 15 // Default value of data-member
  define myData = $myData // A getter
  define setMyData[$newValue] = set $myData = $newValue // A setter
}
```
(4.15)

In code sample (4.15) the method `setMyData` returns a new instance of `MyType`, with the data member `$myData` set to something else. The code sample (4.16) shows the use of a getter and a setter.

```
let $myInstance1 = MyType[],
    $myInstance2 = $myInstance1.setMyData[5],
    $value1 = $myInstance1.myData, // 15
    $value2 = $myInstance2.myData // 5
  in $value1 != $value2 // true
```
(4.16)

The call to `setMyData` does not change the state of the original instance of `MyType`, instead it returns a new instance.

Big-step semantics

The semantics presented in table 4.5 is the transition rule for set expressions.

$$\text{[SET]}\quad \frac{env_C, env_V, env_T \vdash \langle e_1 \rangle \to u_1 \quad \dots \quad env_C, env_V, env_T \vdash \langle e_k \rangle \to u_k}{env_C, env_V, env_T \vdash \langle \texttt{set}\ x_1 = e_1, \dots, x_k = e_k \rangle \to v_1}$$

where $env_C\ \texttt{this} = (t, env'_C, env'_V, v_2)$
and $env''_V = env'_V\,[x_1 \mapsto u_1, \dots, x_k \mapsto u_k]$
and $v_1 = (t, env'_C, env''_V, v_2)$

Table 4.5: *Transition rules for set expressions.*

The transition rule assumes that the current constant environment (env_C) contains a pointer to the current object, `this`. It then returns a copy of that object with a new variable environment, containing the new data.

4.3.7 Operators and calls

Operators are useful for doing calculations, and Junta supports basic mathematical operators and precedence. In order to prevent left-recursion, which makes it possible for us to construct an LL-parser (this was discussed in section 2.4.3), but preserve left-associativity we have created a hierarchy of operators, taking advantage of LL-parsing by putting the operators with highest precedence the lowest in any parse tree that includes them. The grammar for the operators of Junta are described using operator sequences. A sequence is essentially just a list of operations on that particular precedence level. In this way all the precedence levels of Junta are described formally:

$$\begin{aligned}
expression &\to \cdots \mid \texttt{"not"}\ expression \mid lo_sequence \\
lo_sequence &\to eq_sequence\ \{(\ \texttt{"and"} \mid \texttt{"or"}\ )\ eq_sequence\} \\
eq_sequence &\to cm_sequence\ \{(\ \texttt{"=="} \mid \texttt{"!="} \mid \texttt{"is"}\ )\ cm_sequence\} \\
cm_sequence &\to as_sequence\ \{(\ \texttt{"<"} \mid \texttt{">"} \mid \texttt{"<="} \mid \texttt{">="}\ )\ as_sequence\} \\
as_sequence &\to md_sequence\ \{(\ \texttt{"+"} \mid \texttt{"-"}\ )\ md_sequence\} \\
md_sequence &\to negation\ \{(\ \texttt{"*"} \mid \texttt{"/"} \mid \texttt{"\%"}\ )\ negation\} \\
negation &\to element \\
&\mid \texttt{"-"}\ negation \\
element &\to call_sequence\ \{member_access\} \\
member_access &\to \texttt{"."}\ constant\ \{list\} \\
call_sequence &\to atomic\ \{list\}
\end{aligned}$$

In order to completely understand this grammar, we must first take a look at the list of operators ordered by precedence. The precedence of operators is presented in table 4.6.

Level	Operator precedence	
	Operator	Description
1	`f[]`	Function/constructor invocation and list access
2	`r.m r.m[]`	Record member access and member invocation
3	`-`	Unary negation operation
4	`* / %`	Multiplication, division, and modulo
5	`+ -`	Addition and subtraction
6	`< > <= >=`	Comparison operators
7	`== != is`	Equality operators and type checking
8	`and or`	Logical *and* and *or*
9	`not`	Logical *not*
10	`if let set #`	if-, let-, set-, and lambda-expressions

Table 4.6: *The precedence of operators in* JUNTA.

Each precedence level will correspond to a certain rule in the grammar. For instance the fifth precedence level for addition and subtraction is expressed using the $as\_sequence$ rule. Combined with some multiplication an expression making use of the $as\_sequence$ and $md\_sequence$ rules could look like code sample (4.17):

```
2 + 3 * 5 - 3 + 2 / 3
```
(4.17)

The resulting parse tree, using the grammar of JUNTA, could look somewhat like the tree in figure 4.2.

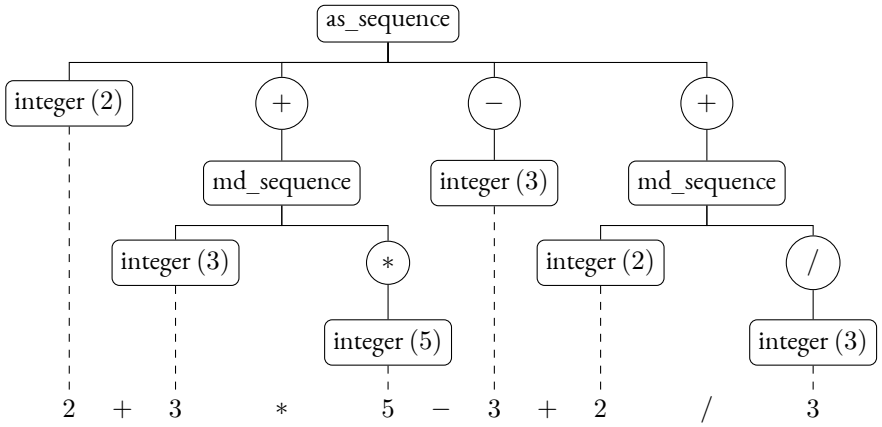


Figure 4.2: *A parse tree for the expression* $2 + 3 * 5 - 3 + 2/3$.

The figure clearly shows the precedence, because the lower nodes will be evaluated before the nodes that are higher in the parse tree. E.g. all of the multiplication and division nodes will be calculated before the additions and subtraction. The circular nodes contain information about each operation in a sequence, i.e. all children of a sequence, except the first one, must have information about the operation. How exactly this is done, is up to the parser. It could also be an attribute on each node, decreasing the number of nodes. The extra information is needed, since for instance addition and subtraction must be on the same precedence level. Making two separate sequences for addition and subtraction would give one operator precedence over the other.

BIG-STEP SEMANTICS

The semantics presented in table 4.7 is the transition rule for function calls. Big-step semantics for the others are left out, as they are mostly trivial.

$$[\text{CALL}_{\text{FUN}}] \quad \frac{\begin{array}{l} env_C, env_V, env_T \vdash \langle e \rangle \rightarrow v_2 \\ env_C, env_V, env_T \vdash \langle i \rangle \rightarrow v_3 \\ env'_C, env''_V, env_T \vdash \langle e' \rangle \rightarrow v_1 \end{array}}{env_C, env_V, env_T \vdash \langle e\ i\ \rangle \rightarrow v_1}$$

where $v_2 = (g, e', env'_V, env'_C)$
and $v_3 = (l, elem)$
and $env''_V = [x_1 \mapsto elem\ 1, \ldots, x_n \mapsto elem\ n]$

Table 4.7: *Transition rules for function calls.*

In this rule the expression e is evaluated to a function value v_2, which has its own variable and constant environments, as the static scope rules. The variable environment is then updated with the actual parameters assigned to the formal parameters, after which the expression, contained within the function value, is evaluated.

The semantics presented in table 4.8 is the transition rule for member access, also known as dot-notation.

$$[\text{MEMBER}_{\text{ACCESS}}] \quad \frac{env_C, env_V, env_T \vdash \langle e \rangle \rightarrow v_1}{env_C, env_V, env_T \vdash \langle e.C \rangle \rightarrow v_3}$$

where $v_2 = (t, env'_C, env'_V, v_2)$
and $env'_C\ C = v_3$

Table 4.8: *Transition rules for member access.*

A member access is as simple as evaluating the left-side of the object first (the constant), and then accessing the evaluated constant in that objects constant environment.

Valid operands

In the following paragraphs we present the valid operands for the operators of Junta. These will be grouped in the following categories: boolean, comparison, integer, string, list, and direction and coordinate operators.

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 to compare two values, and always returns a boolean value:

- `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.
- `* == * → Boolean`
  Returns true if the left operand is equal to the right one.
- `* != * → Boolean`
  Returns true if the left operand is not equal to the right one.
- `* is Type → Boolean`
  Returns true if the type of the first operand is equal to or inherits from the type operand.

Integer operators The following operations are possible on integers:

- `- Integer → Integer`
  Integer negation.
- `Integer + Integer → Integer`
  Integer addition.
- `Integer - Integer → Integer`
  Integer subtraction.
- `Integer * Integer → Integer`
  Integer multiplication.
- `Integer / Integer → Integer`
  Integer division.
- `Integer % Integer → Integer`
  Integer modulo operation.

String operators It is possible to concatenate strings:

- `String + String → String`
  Returns the concatenation of two strings.
- `String + * → String`
  `* + String → String`
  Returns the concatenation of a string and the string-representation of another type

List operators Some operators are available for list values as well:

- `List + List → List`
  Returns a list containing all elements from the first list followed by all elements from the second list.
- `List - List → List`
  Returns a list containing all the elements from the first list that do not exist in the second list.
- `List + * → List`
  Appends any element on to the end of a list, and returns the resulting list.
- `List - * →`
  Returns a list containing the elements that do not match the right operand.
- `* + List → List`
  Prepends any element on to the start of a list, and returns the resulting list.

Direction and coordinate operators The following operators can manipulate directions and coordinates:

- `Direction + Direction → Direction`
  Add a direction (vector) to another direction.
- `Direction - Direction → Direction`
  Subtract a direction from another direction.

- `Direction + Coordinate` → `Coordinate`
  Add a coordinate to a direction.
- `- Direction` → `Direction`
  Negate a direction.
- `Coordinate - Coordinate` → `Direction`
  Returns the distance between two coordinates as a direction.
- `Coordinate + Direction` → `Coordinate`
  Add a direction to a coordinate.
- `Coordinate - Direction` → `Coordinate`
  Subtract a direction from a coordinate.

For instance adding the directions `n` and `e` produces a direction equivalent with the direction `ne`. Adding a coordinate and direction (and vice versa) gives a coordinate. As an example, `A2 + e` gives `B2`. More information about the coordinate and direction types is available in section 4.6.1.

4.4 Definitions

In this section we present how programs written in Junta can be structured with definitions of constants and types.

We begin by presenting what a program can consist of and then we further specify how these different definitions are built. We present constant definitions followed by type definitions. We provide big-step semantics for type definitions in section 4.4.3.

4.4.1 Program structure

The outermost layer of a Junta program is a list of definitions:

$$\begin{array}{rcl} program & \rightarrow & \{definition\} \\ definition & \rightarrow & constant\_def \\ & | & type\_def \end{array}$$

A definition is either a constant definition or a type definition. So, when a program has been run, we are left with a symbol table full of types and constants.

As the above grammar shows, an empty program is valid in Junta, because it is possible to have zero, one, or more definitions in the outermost layer of the structuring of programs.

4.4.2 Constant definitions

Functions are also constants and are defined with the following definition:

$$constant\_def \quad \rightarrow \quad \texttt{"define"}\ constant\ [\ varlist\ ]\ \texttt{"="}\ \ expression$$

A function definition needs a list of formal parameters. Formal parameters which are represented as $varlists$ are described with the following grammar:

$$\begin{array}{rcl} varlist & \rightarrow & \texttt{"["}\ [\ variable\ \{\texttt{","}\ variable\}\ [\ \texttt{","}\ vars\ ]\ |\ vars\ ]\ \texttt{"]"} \\ vars & \rightarrow & \texttt{"..."}\ variable \end{array}$$

Creating constants and functions outside of type definitions adds them to the global scope, meaning that they are essentially accessible from anywhere in the program (provided that they are not hidden by a constant within a type).

An example of a global function is as the following implementation of a function for computing the greatest of two numbers, presented in code sample (4.18).

```
define max[$a, $b] = if $a > $b then $a else $b
```
(4.18)

Essentially this creates a constant in the global scope named `max`, which when used, returns a value of type `Function`. Since function values can also be created with lambda expressions (as described in section 4.3.5), the following constant definition is equivalent to code sample (4.18):

```
define max = #[$a, $b] => if $a > $b then $a else $b
```
(4.19)

This equivalence only holds for global constants, since type constants/methods are a bit more special as described in section 4.4.3.

Constants and functions are useful for putting frequently used expressions or values in one place.

The non-terminal *vars* is used for variadic functions, another feature of JUNTA. Variadic functions are functions with indefinite arity, meaning they will accept any number of actual parameters. In JUNTA this is supported for both lambda expressions and functions. The code sample (4.20) is an example of two variadic functions.

```
define last[... $args] = $args[-1]
define myMap[$func, ... $list] = $list.map[$func]
```
(4.20)

The use of the `"..."` terminal marks that the following variable represents a list that holds all additional parameters passed to the function. In code sample (4.20), calling the function `last` with no parameters, is possible and results in the formal parameter `$args` holding the value of an empty list, `[]`. Additionally, `last` can be called with any number of parameters, which will then be appended to the list in `$args`. In the second function, `myMap`, at least one actual parameter must be provided (since the variadic parameter is the second one). But other than that, the parameter passing works in the same way as with `last`, in that the function accepts any number of parameters greater than or equal to 1. Some uses and results of the two functions are shown in code sample (4.21) and code sample (4.22).

```
last["test", 15, true, A15] // returns A15
```
(4.21)

```
myMap[#[$a] => $a * 10, 0, 1, 2, 3] // returns [0, 10, 20, 30]
```
(4.22)

One limit is that the variadic parameter (the one after `"..."`) must be the last one in the list of formal parameters, and there can only be one. This is expressed in the grammar.

SCOPE RULES FOR FUNCTIONS AND CONSTANTS

Consider the `max` function in code sample (4.18). Its formal parameters `$a` and `$b` only exist and are only available within that function. The example in code sample (4.23) shows a call of the `max` function, which was presented in code sample (4.19).

```
max[5, 23]
```
(4.23)

When called with the actual parameters 5 and 23, a new scope is created and the actual parameters are assigned to the formal parameters `$a` and `$b`, respectively. The body of the function (the if expression) is then evaluated and the result is returned. When returning, the variables `$a` and `$b` cease to exist.

4.4.3 Type definitions

As described in the introduction to this chapter, types are a central part of Junta and being able to define custom user types is essential when creating board games. The following grammar rules present how type definitions work in Junta followed by the associated definitions which can be used within a type definition.

$$
\begin{aligned}
\mathit{type\_def} &\rightarrow \texttt{"type"}\ \mathit{type}\ \mathit{varlist}\ [\ \texttt{"extends"}\ \mathit{type}\ \mathit{list}\ ]\ [\ \mathit{type\_body}\ ] \\
\mathit{type\_body} &\rightarrow \texttt{"\{"}\ \{\mathit{member\_def}\}\ \texttt{"\}"} \\
\mathit{member\_def} &\rightarrow \mathit{abstract\_def} \\
&\mid \mathit{constant\_def} \\
&\mid \mathit{data\_def} \\
\mathit{abstract\_def} &\rightarrow \texttt{"define"}\ \texttt{"abstract"}\ \mathit{constant}\ [\ \mathit{varlist}\ ] \\
\mathit{data\_def} &\rightarrow \texttt{"data"}\ \mathit{variable}\ \texttt{"="}\ \mathit{expression}
\end{aligned}
$$

The simplest type that can be created is a type such as the example in code sample (4.24).

```
type A[]
```
(4.24)

The type `A` is a type without any data, constructor parameters, constants, or parent type. This type is truly useless, since it has no identity or behaviour. It can however be instantiated, and instances of it can be compared using the `==`, `!=`, and `is` operators. But since the type has no identity in any way, all instances will be equal as presented in code sample (4.25).

```
A[] == A[] // true
typeOf[A[]] == A // true
A[] is A // true
```
(4.25)

The constructor

One way to add identity to objects, is with the constructor. In the previous code sample (4.24) of a very simple type, the constructor takes no parameters (the empty parameter list `[]` after the type name). If we were to add some formal parameters to the type definition, it could look like the code sample (4.26).

```
type A[$b, $c]
```
(4.26)

Now in order to instantiate `A`, we must provide the constructor with two parameters as presented in code sample (4.27).

```
A[12, 23] == A[12, 23] // true
A[12, 23] != A[10, 17] // true
```
(4.27)

Notice how in the first line, the two objects are equal to each other, while in the second line they are not. This means that we have successfully added identity to the `A` type.

The constants

Constants within types are defined in the same way as constants outside of types. The difference is that constants defined within a type can only be accessed within that type (and inheriting types) or by using the dot-notation outside of the type. This is illustrated in code sample (4.28).

```
type A[$b, $c] {
  define b = $b + 1
  define calculate[$d] = b + $c + $d
}
```
(4.28)

In code sample (4.28), two constants are defined within type `A`: `b` and `calculate`. `b` is a simple constant holding the value of `$b`, the constructor parameter, plus one. It can be seen as a getter, since it makes the value of `$b` visible to the outside. `calculate` is a method that returns the sum of some numbers. In order to call the constants contained within a type, we use the dot-notation on an object of that type illustrated in code sample (4.29).

```
let $obj = A[5, 3]
  in $obj.calulate[3] // (5 + 1) + 3 + 3 = 12
```
(4.29)

The variable `$obj` is assigned an instance of the `A` type. Using the dot-notation, the method `calculate` is called on the object.

The data

Another way to add identity to objects, is to add data fields to the type. Unlike constants, data fields contain private semi-mutable data. In the strictest sense, the data is still immutable, but by using the `set` keyword, it is possible to clone the current object, and return a new one with the selected data fields set to new values. The example in code sample (4.30) shows a new version of type `A`, with a data field.

```
type A[] {
  data $att = 15
  define att = $att
  define setAtt[$val] = set $att = $val
}
```
(4.30)

In this example, we define the data field `$att` with the default value of 15. Since data fields are not accessible from outside of the type, we must define a getter, the constant `att`, in order to make the value visible. Using the `set` keyword, we can also define at setter, the `setAtt` method, which returns a new instance of `A` with `$att` set to whatever parameter `setAtt` was called with.

The example in code sample (4.31) shows the use of this setter, to create a clone of an instance of `A`.

```
let $obj1 = A[],
  $obj2 = $obj1.setAtt[2]
  in [
    $obj1.att, // 15
    $obj2.att, // 2
    $obj1 != $obj2 // true
  ]
```
(4.31)

This time we create an instance of `A`, and then call `setAtt` on that instance, in order to get a new instance of `A` with `$att` set to 2. After that, the values are accessed using the getter. Note that again the two objects are not equal, since the value of `$obj2` is different than `$obj1`.

The super type

In JUNTA single inheritance is possible using the `extends` keyword. When extending another type, that type's constructor must be invoked with some parameters. The following complicated example illustrated in code sample (4.32) shows how single inheritance can be used.

```
// Abstract type because of abstract members
type A[$a] {
  define abstract constantA
  define methodA[$arg] = constantA + $arg / $a
}
// Abstract type because of unimplemented members (constantA)
type B[$b, $c] extends A[$b + $c] {
  // Overrides method in A (must have same arity)
  define methodA[$arg] = super.methodA[$arg + 2]
}
// Concrete type, implements abstract member from A
type C[] extends B[2, 4] {
  // Overrides member in A (must be a constant)
  define constantA = 5
  define anotherMethod[$arg] = methodA[$arg / 2]
}

define use = C[].anotherMethod[20] // 7 = 5 + (20 / 2 + 2) / (2 + 4)
```

(4.32)

Junta doesn't have an `abstract` keyword for type definitions, only for constant/function definitions. A type is implicitly marked as abstract if it has unimplemented abstract members. In example presented in code sample (4.32), the type `A` is abstract since its member `constantA` is abstract. Likewise, the type `B` is abstract because it extends `A`, but doesn't implement the abstract member of `A`. The type `C` on the other hand is not abstract, since it implements the abstract constant. Abstract types can't be constructed, albeit the constructor for an abstract type has to be used when extending the type (after the `extends` keyword).

When instantiating type `C`, several things happen. First `C`'s parent is instantiated using the actual parameters 2 and 4. Then `B`'s parent is instantiated using the value of `$b + $c`. It is worth noting that, when calling the parent constructor, it is not possible to access constants or data from within the type. In figure 4.3 a visualisation of the scopes of a type definition is shown.

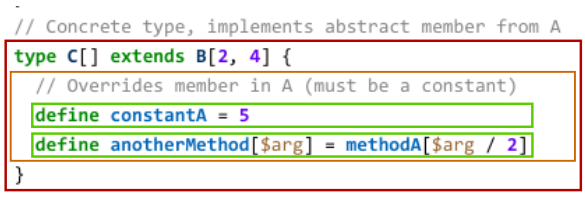


Figure 4.3: *The scopes of a type definition. Each scope is marked with a rectangle. A scope contained within another scope has access to the variables of the parent scope.*

Overriding

Junta supports overriding and abstract constants/methods. Any constant in the parents can be overridden in the subtype. It is however required that the overriding constant has the same signature as the original. Since Junta is dynamically typed, the only constraints are:

1. If the original member is a constant (no formal parameter list after the name), then the overriding member must be a constant as well
2. If the original member is a method with n formal parameters, then the overriding method must have n formal parameters as well

So, unlike in the global scope, there is a small difference between constants and functions (methods). For abstract members, the same constraints hold.

Big-step semantics

The semantics presented in table 4.9 are the transition rules for type definitions. These type definitions have some optional arguments which correspond with the written grammar for these definitions, and this is why there are four transition rules described.

[TYPEDEF]	$\dfrac{env_C \vdash \langle D_G, env_T[T \mapsto (T, g, \varepsilon, \varepsilon, \varepsilon)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T\ g\ D_G,\ env_T\rangle \rightarrow env'_T}$
[TYPEDEF$_{\text{BODY}}$]	$\dfrac{env_C \vdash \langle D_G, env_T[T \mapsto (T, g, D_M, \varepsilon, \varepsilon)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T\ g\ \{D_M\}\ D_G,\ env_T\rangle \rightarrow env'_T}$
[TYPEDEF$_{\text{EXTEND}}$]	$\dfrac{env_C \vdash \langle D_G, env_T[T_1 \mapsto (T_1, X, \varepsilon, i, T_2)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T_1\ g\ \texttt{extends}\ T_2\ L\ D_G,\ env_T\rangle \rightarrow env'_T}$
[TYPEDEF$_{\text{EXTEND-BODY}}$]	$\dfrac{env_C \vdash \langle D_G, env_T[T_1 \mapsto (T_1, g, D_M, i, T_2)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T_1\ g\ \texttt{extends}\ T_2\ i\ \{D_M\}\ D_G,\ env_T\rangle \rightarrow env'_T}$

Table 4.9: *Transition rules for type definitions.*

In the premises of the rules we present a 5-tuple, a **TypeValue** where env_T is updated according to the rule. In three of the four 5-tuples we include the symbol ε, which denotes that the given position of the symbol is an empty slot. This is again due to the fact that we have some optional arguments.

The 5-tuple is ordered as follows:

1. $\mathbf{T_1}$ – current type
2. $\mathbf{X}$ – current type's formal parameters
3. $\mathbf{D_M}$ – member definitions
4. $\mathbf{L}$ – super type's parameters
5. $\mathbf{T_2}$ – super type

Throughout the transition rules we use the 5-tuple to update the type environment, env_T.

4.5 Patterns

This section covers how to use patterns. The operators of a pattern looks like and behaves a little like regular expressions. This grammar shows how a pattern is constructed:

$$
\begin{array}{rcl}
\textit{pattern} & \rightarrow & \textit{pattern\_expr}\ \{\textit{pattern\_expr}\} \\
\textit{pattern\_expr} & \rightarrow & \textit{pattern\_val}\ [\ \texttt{"*"} \mid \texttt{"?"} \mid \texttt{"+"}\ ] \\
 & \mid & \textit{pattern\_val}\ \texttt{"|"}\ \textit{pattern\_expr} \\
\textit{pattern\_val} & \rightarrow & \textit{direction} \\
 & \mid & \textit{variable} \\
 & \mid & \textit{pattern\_check} \\
 & \mid & \texttt{"!"}\ \textit{pattern\_check} \\
 & \mid & \texttt{"("}\ \textit{pattern}\ \texttt{")"}\ [\ \textit{integer}\ ]
\end{array}
$$

pattern_check → `"friend"`
| `"foe"`
| `"empty"`
| `"this"`
| *type*

A pattern is checked on a particular square, and returns either true or false. An example of a pattern is `/n n e empty/`. This pattern can be checked on the board seen in figure 4.4 on the field `A1`. The pattern says "move one square north, move one square north, move one square east, check if square is empty". This means that the square `B3` will be checked if it is empty. Since the square is occupied by a piece, the pattern will return false if checked on `A1`. However, the same pattern matched on `C1` will return true since the square at `D3` is empty.

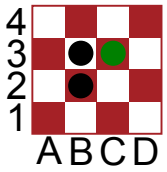


Figure 4.4: *A simple* 4×4 *board with* 3 *pieces.*

With this basic introduction to a simple pattern matching, table 4.10 briefly describes how the different pattern operators work. For each operator an example of the use in a context is given in section 4.5.1. Note that the description of patterns assumes a minor understanding of regular expressions, see [8][16].

Pattern operator	Explanation
`empty`	current square contains no pieces
`n`	north
`e`	east
`s`	south
`w`	west
`*`	zero-to-many times
`+`	one-to-many times
`?`	zero-or-one time
`|`	or
`!`	not
(*pattern*)	encapsulation
`friend`	current square contains at least one friendly piece of the current player
`foe`	current square contains at least one enemy piece of the current player
`type`	the current square is of the given type, or a piece of the given type is residing on the current square

Table 4.10: *Pattern operators.*

4.5.1 Pattern examples

All these examples of pattern matching are performed on the board and pieces seen in figure 4.4. For each operator two examples of a pattern matching on a particular square is given. The first check returns true and the second check returns false.

1. On `A1`: the pattern matching `/empty/` returns true because `A1` is empty
2. On `B2`: the pattern matching `/empty/` returns false because `B2` is not empty
3. On `A1`: the pattern matching `/n empty/` returns true because `A2` is empty

4. On B1: the pattern matching /n empty/ returns false because B2 is not empty
5. On C3: the pattern mathcing /e empty/ returns true because C4 is empty
6. On C4: the pattern matching /e empty/ returns false because C5 is out of board
7. On A1: the pattern matching /n* n e empty/ returns true because moving north 2 times to A3, then north and east hits an empty square on B4

Notice that the * operator causes many branches to be made. The previous pattern /n* n e empty/ done on A1 has a branch checking /n e empty/. The branch dies because B2 is not empty. If just one branch survives, the pattern matching returns true. In the example, the only branch surviving is the n n n e empty branch. The same rules for branching counts for the +, ?, and | operators. When a branch moves out of board it dies immediately.

8. On C3: the pattern matching /n* s s !empty/ returns false because neither A1 nor A2 contains a piece
9. On A1: the pattern matching /n+ e empty/ returns true only because B4 is empty. Notice how this pattern matching is equivalent to the pattern matching in bullet 7
10. On B1: the pattern matching /n+ empty e !empty/ returns false because B4 is the only empty square north of B1 and C4 is empty
11. On B3: the pattern matching /s? e empty/ returns true only because C2 is empty
12. On C2: the pattern matching /n? w empty/ returns false because neither B2 nor B3 is empty
13. On A2: the pattern matching /(n | e) empty/ returns true only because A3 is empty
14. On C2: the pattern matching /e | w empty/ returns false because neither B2 nor C3 is empty
15. On B1: the pattern matching /(n n | e e) empty/ returns true only because D1 is empty
16. On A1: the pattern matching /(n w)* empty/ returns true both because A1 is empty and because D4 is empty

Pattern keywords

The friend keyword is evaluated based on the current player. Suppose that we have a player called Green, who owns the green piece. If it is Green's turn to move, any branch of a pattern matching will return false whenever it meets a keyword friend on a square that does not contain any of Green's pieces. On B2, the pattern matching /n | e | (n e) friend/ will return true if it is Green's turn, since C3 contains a friendly piece of Green. On C3, the pattern matching /(e | w)+/ will return false if it is Green's turn, since no square containing a green piece can be reached by going east or west from C3 one or more times.

The foe keyword does the opposite of friend. It makes a branch continue only if its current square contains at least one piece not owned by the current player's turn.

Just like foe and friend, the name of a piece or square type defined in a Junta game can also be used. E.g. the type identifier White can be used if a piece or square type with the name has been defined. In such a case, a branch survives only if its current square is of type White or if the square contains a piece of type White.

A pattern can also match if a specific piece exists on a specific square. This is done using the this keyword. Before this matching can be achieved, the pattern must be matched regarding to a specific piece. Suppose that we on A3 make the pattern matching /empty e this/. If this matching is done in relation to the black piece on B3 it returns true. However, in relation to the black piece on B2, the pattern matching returns false.

To understand how this function works and why this is useful, consider the board in figure 4.4. If we for any piece specify that it can move to a square for which the pattern matching /(n | s) empty/ is true, this means that it can move to any square except {**B1**, **B4**, **C4**}. These squares do not have an empty square north or east from them. Recall that a branch going out of board dies. However, the pattern matching /empty (n | s) this/ will in relation to the green piece return true only when matched on the squares {**C2**, **C4**}. This can be used to specify that a piece can move to an empty square one north or one south from its current square.

4.6 Predefined types and constants

In this section we introduce the standard environment of Junta in section 4.6.1, followed by the game environment in section 4.6.2.

In order to write programs in a programming language, it is often necessary to make use of built-in functions and types. Junta provides a number of built-in functions and constants, as well as simple types for representing values such as integers and strings. Junta also provides a type hierarchy designed for implementing and expressing board games. This will in many cases make it much easier for programmars to program in Junta since they don't have to implement the functionality which the built-ins already provides.

Since Junta doesn't have a module, package, or name space system, the distinction between a standard- and game environment doesn't actually exist in the language, and all types and constants exist in the same global name space. The distinction between the two is merely formal and based on the sort of types and functionality that each provide.

4.6.1 Standard environment

We now introduce the standard environment of Junta. The standard environment provides the simple types, such as integers, strings, boolean values, etc. and their related functions and constants for working with the these.

The following global constants are available:

- `typeOf[$value : *] : Type`
  A function that returns the type of any value.
- `union[$list : List, ..., $lists : List] : List`
  A function that returns the union of a number of lists.
- `true : Boolean`
  The boolean true value.
- `false : Boolean`
  The boolean false value.

Integer

- `Integer[$integer : Integer]`
  The standard environment provides the `Integer` type, which is implemented as Java's primitive data type, **Integer**. That is, it is a 32-bit signed two's complement integer. When the interpreter detects a numeral, it returns an integer value object. If for instance the numeral exceeds the highest possible value, a **TypeError** is thrown.

Boolean

- `Boolean[$boolean : Boolean]`
  The standard environment provides the `Boolean` type, which is implemented as Java's primitive data type, **Boolean**. That is, it only has two possible values: true and false. Even though the data type represents only one bit of information, according to the Java documentation, the "size" isn't precisely defined.

String

- `String[$string : String]`
  The standard environment provides the `String` type, which is implemented as Java's data type, **String**. That is, it may contain any unicode (UTF-16) characters. Though it is not possible to write unicode characters of the form "\uXXXX" as in Java (for instance "\u0108", which is the capital C with circumflex, Ĉ). The `String` type contains one built-in constant:
  - `size : Integer`
    The `size` constant returns the number of characters in the string, which is an integer value. For example `$testString.size` = 11.

List

- `List[$list : List]`
  The standard environment provides the `List` type. A list object can contain a mix of any type: strings, integers, other lists, game objects, etc. This has both advantages and disadvantages. It increases the orthogonality of the programming language but it increases the risk of getting errors, which don't show until at run time. The `List` type is similar to the **ArrayList** of Java and it is resizeable, which means that types can be added to the `List`. The type comes with a number of built-in constants and functions.
- `size : Integer`
  The `size` constant returns the number of elements in the list, which is an integer value. E.g. `[$testString, 2, 4].size` $= 3$.
- `sort[$comparator : Function] : List`
  The `sort` function sorts a list using a function that must take two parameters as input and return an integer value. For instance, the following will sort a list in acsending order: `[1, 6, 2, 5, 4, 3].sort[#[$a, $b] => if $a > $b then 1 else if $a == $b then 0 else -1]`. The result will be `[1, 2, 3, 4, 5, 6]`. As shorter expression for sorting in ascending order is the following lambda expressions which gives the same result as the previous example with the if expression: `[#[$a,$b] => $a - $b]`
- `map[$mapper : Function] : List`
  The `map` function maps each element of the list with a function of style `#[$a] => $a`. The function must take one parameter. For example `[1, 2, 3, 4, 5, 6].map[#[$a] => $a + 1]` will return the list: `[2, 3, 4, 5, 6, 7]`.
- `filter[$filter : Function] : List`
  The `filter` function filters a list by feeding it with a function of style `#[$a] => $a >= 5`, and returns a list with only the elements which comply with the function. The function fed to the `filter` function must take one parameter and return a boolean value. For example `[1, 2, 3, 4, 5, 6].filter[#[$a] => $a >= 5]` will return `[5, 6]`.

Direction

- `Direction[$direction : Direction]`
  The standard environment provides the `Direction` type, which can be compared to a vector. There are eight different directions: `n` (north), `s` (south), `w` (west), `e` (east), `nw`, `ne`, `sw`, `se`. The type consist of an x value and a y value. For example `n` has value $y = 1$ and $x = 0$, `s` has value $y = -1$ and $x = 0$, `w` has value $y = 0$ and $x = -1$, etc. The `Direction` type is meant as a practical tool for use in patterns.

Coordinate

- `Coordinate[$coordinate : Coordinate]`
  The standard environment provides the `Coordinate` type. This type is closely related to the `Direction` type in the way that it also consist of a x and y value. When the interpreter detects a number of capital letters followed by a one or more numerals it returns a coordinate value object. Examples of coordinate values are `A1`, `Z99` and `ABCD1234`. The coordinate value `A1` corresponds to the $x = 1$ and $y = 1$, which is the bottom-left square on a board. The coordinate type is ment as a practical tool to specify squares on a grid-formed board. Coordinate values must be positive, as negative x and y values make no sense representing coordinates off of the board.

Type

- `Type[$type : Type]`
  The `Type` type is the meta type of JUNTA, all types, including `Type` itself, are instance of this type. The constructor for `Type` accepts one parameter, a type, an essentially just returns that type. This can be used for casting instances of subtypes of `Type`.
- `isSubtypeOf[$other : Type] : Boolean`
  Returns true if this type is a subtype of the other type. False otherwise.
- `isSupertypeOf[$other : Type] : Boolean`
  Returns true if this type is a super type of the other type. False otherwise.

Function

- `Function[$function : Function]`
  This type represents a callable function. Instances of this type are returned by lambda expressions and functions/methods are referred to by name.
- `call[$parameters : List] : *`
  Calls the function with the specified parameter list. This can be used instead of the `func[]` method, for instance in order to create a list of actual parameters dynamically.

Pattern

- `Pattern[$pattern : Pattern]`
  This type represents a pattern used for pattern matching on a `Game` object.

4.6.2 Game environment

The game environment provides a class hierarchy for describing a board game in an object-oriented manner. In the game environment the following global functions are available:

- `addAction[$piece : Piece, $squares : List] : List`
  A function that returns a list of `AddAction` types to where it is possible to add a piece (`$piece`). The functions take two parameters. The first parameter contains information on which type of piece the actions applies to. The second parameter is the list of squares where the type of piece can be added to. In the code sample (4.1) in the beginning of the chapter, `addAction` is used in the following way:

  ```
  addAction[$pieceType[this], $gameState.board.emptySquares]
  ```

  Here `addAction` returns a list of empty squares to where it is possible to add a piece of the type `this`, which in this case was either a crosses piece or noughts piece depending on whose turn it is.
- `moveAction[$piece : Piece, $squares : List] : List`
  `moveAction` works like `addAction`, but instead of returning a `List` of `AddAction` types it returns a `List` of `MoveAction` types.

Game

The `Game` type contains all information needed to describe a board game.

- `Game[$title : String]`
  Creates an instance of the `Game` with a game title of `$title`, `board` set to `initialBoard`, and `currentPlayer` set to `turnOrder[0]`.
- `players : List`
  List of all `Player` objects that are a part of this game.
- `currentPlayer : Player`
  The `Player` type from `players` which currently have the turn.
- `turnOrder : List`
  The order of `Player` types which determines in which order each `Player` from `players` has their turn.
- `initialBoard : Board`
  The value of `board` at the beginning of each game.
- `board : Board`
  The current state of a `Board` for this game.
- `title : String`
  The title of the game which users can identify the game with.
- `description : String`
  A short explanation of the game and/or its rules.

- `matchSquare[ $position : Coordinate, $pattern : Pattern ] : Boolean`
  Is true if `$pattern` is valid for `$position`.
- `matchSquares[ $positions : List, $pattern : Pattern ] : Boolean`
  Is true if and only if all `Coordinates` in `$positions` are true for `matchSquare` with `$pattern`.
- `findSquares[ $pattern : Pattern ] : List`
  `List` of all `Squares` where its `position` matches `$pattern`.
- `findSquaresIn[ $positions : List, $pattern : Pattern ] : List`
  `List` of `Squares` where its `position` matches `$pattern`, but only `Squares` where `Coordinate` exists in `$positions`.
- `history : List`
  `List` of all applied `Action` types.
- `applyAction[ $action : Action ] : Game`
  A `Game` where `board` has been updated according to `$action` and where `$action` is appended to `history`.
- `undoAction[] : Game`
  A `Game` where `board` has been reset to its state before the last action in the history was applied and with `history` updated accordingly.
- `setHistory[ $history : List ] : Game`
  A `Game` where `history` is equal to `$history`.
- `setBoard[ $board : GridBoard ] : Game`
  A `Game` where `board` is equal to `$board`.
- `setCurrentPlayer[ $i : Integer ] : Game`
  A `Game` where `currentPlayer` is `turnOrder[$i]`.
- `nextTurn[] : Game`
  The `Player` which has the turn after `currentPlayer`.

Board

- `Board[]`
  A `Board` with no `Pieces`.
- `pieces : List`
  A `List` containing all `Pieces` associated with the `Board`.
- `setPieces[ $pieces : List ] : Board`
  A `Board` where `pieces` is equal to `$pieces`.

GridBoard

`GridBoard extends Board` to provide an easy way to describe rectangular `Boards`.

- `GridBoard[ $width : Integer, $height : Integer ]`
  A `GridBoard` with `width` and `height` being set to `$width` and `$height`, respectively.
- `width : Integer`
  The width of the rectangular `Board`.
- `height : Integer`
  The height of the rectangular `Board`.
- `squares : List`
  A `List` of all associated `Squares`.
- `setSqaures[ $squares : List ] : GridBoard`
  A `GridBoard` where `squares` is equal to `$squares`.
- `addPiece[ $piece : Piece, $position : Coordinate ] : GridBoard`
  A `GridBoard` where `$piece` is appended to `pieces` and added to the `Square` at `$position`.
- `addPieces[ $piece : Piece, $positions : List ] : GridBoard`
  A `GridBoard` where `$piece` is appended to `pieces` and added to all the `Squares` at any of `$positions`.
- `removePiece[ $piece : Piece ] : GridBoard`
  A `GridBoard` where `$piece` is off-board.

- `movePiece[ $piece : Piece, $position : Coordinate ] : GridBoard`
  A `GridBoard` where `$piece` (which is already contained in `pieces`) is `onBoard` and is only included in one `Square`'s `pieces`.
- `squareAt[ $position : Coordinate ] : Square`
  The `Square` at `$position` in the rectangular grid of `GridBoard`.
- `setSqauresAt[ $square : Square, $position : List ] : Square`
  A `GridBoard` where `squareAt[ $position ]` is equal to `$square`.
- `isFull : Boolean`
  Is true if `emptySquares.size` $= 0$.
- `emptySquares : List`
  A `List` with `Square`s from `squares` where `isEmpty` is false.
- `squareTypes : List`
  A `List` with default `Square`s which will be used to create a checkered pattern of `Square`s in the grid of `Square`s.

Square

`Square` describes a position on the `Board` where zero-to-many `Piece`s can be placed.

- `Square[]`
  `Square` with no `Piece`s.
- `position : Coordinate`
  `Coordinate` describing the position on a `GridBoard`.
- `pieces : List`
  A `List` with `Piece`s located on this `Square`.
- `addPiece[ $piece : Piece ] : Square`
  A `Square` where `$piece` is appended to `pieces`.
- `removePiece[ $piece : Piece ] : Square`
  A `Square` where `$piece` is not contained in `pieces`.
- `setPieces[ $pieces : List ] : Square`
  A `Square` where `pieces` is equal to `$pieces`.
- `image : String`
  Path to an image file used for visualising the `Square`.
- `isOccupied : Boolean`
  Is true if `pieces.size` > 0.
- `isEmpty : Boolean`
  Is true if `pieces.size` $= 0$.
- `setPosition[ $position : Coordinate ] : Square`
  A `Square` where `position` is equal to `$position`.

Piece

`Piece` describes an item associated to a `Player` which the `Player` can manipulate in order to progress the game.

- `Piece[ $owner : Player ]`
  `Piece` with `owner` set to `$owner`.
- `owner : Player`
  `Player` which owns this `Piece`.
- `image : String`
  Path to an image file used for visualising the `Piece`.
- `position : Coordinate`
  `Coordinate` for the `Square` this `Piece` is located on.
- `move[ $position : Coordinate ] : Piece`
  A `Piece` with `position` set to `$position` and `onBoard` set to true.

- `remove[] : Piece`
  A `Piece` where `position` is invalid and `onBoard` is false.
- `onBoard : Boolean`
  Is true if `Piece` is on the `GridBoard`.
- `actions[ $game : Game ] : List`
  A `List` of possible `Actions` the `Piece` can make on its `owner`'s turn.

Player

- `Player[ $name : String ]`
  `Player` with `name` set to `$name`
- `name : String`
  The name of the `Player`.
- `winCondition[ $game : Game ] : Boolean`
  Is true if the `Player` has won at the end of this turn.
- `tieCondition[ $game : Game ] : Boolean`
  Is true if the game ended without a winner.
- `actions[ $game : Game ] : List`
  A `List` of `Actions` that the `Player` can do during his turn.

Action

- `Action[]`
  Empty `Action`.

UnitAction

`UnitAction extends Action` to provide a basic change to be performed on `Game`.

- `UnitAction[ $piece : Piece ]`
  A `UnitAction` with `piece` set to `$piece`.
- `piece : Piece`
  The `Piece` this `UnitAction` affects.

AddAction

`AddAction extends Action` to add a `Piece` to a `Game`.

- `AddAction[ $piece : Piece, $to : Square ]`
  An `AddAction` which adds `$piece` to `$to`.
- `to : Square`
  `Square` to add `piece` to.

RemoveAction

`RemoveAction extends Action` to remove a `Piece` from a `Game`.

- `RemoveAction[ $piece : Piece ]`
  A `RemoveAction` which removes `$piece`.

MoveAction

`MoveAction extends Action` to move a `Piece` to another `Square`.

- `MoveAction[ $piece : Piece, $to : Square ]`
  A `MoveAction` which moves `$piece` to `$to`.
- `to : Square`
  `Square` to add `piece` to.

ActionSequence

`ActionSequence extends Action` to provide a sequence of `UnitAction`s to be performed in order.

- `ActionSeqence[ ...$actions : UnitAction ]`
  `ActionSequence` with `actions` set to `[ ...$actions ]`.
- `actions : List`
  A `List` of `UnitAction` to be performed in order.
- `addAction[ $action : UnitAction ] : ActionSequence`
  A `ActionSequence` where `$action` is appended to `actions`.

TestCase

An abstract type for unit testing within a Junta program. It doesn't provide any methods or constants. Its only purpose is to make it possible to identify unit testing types. If a type extends `TestCase`, then each constant within that type is evaluated, and if a `Boolean` true value isn't returned, then the test is considered unsuccessful.

Chapter 5

Implementation

In the following chapter we present how the different components of JUNTA are implemented. In section 5.1 the scanner implementation is specified and examples are given of how the tokens are identified. In section 5.2 we present the parser of JUNTA. We explain how it's constructed and how it works. Furthermore, we present a parser and a scanner generated by SableCC and the reason why we did not proceed with this approach. In section 5.3 our abstract node types are explained and illustrated. In section 5.4 we present the scope checker of JUNTA. Here the visitor pattern and the use of it is also introduced. In section 5.5 the interpreter is presented. Here we explain how patterns are implemented which is a feature used in most programs written in JUNTA and we introduce our implementation of operators and values. In section 5.6 we present how error handling is implemented. In section 5.7 we explain the Game Abstraction Layer which is the layer that binds the interpreter and the simulator together and makes them able to communicate with each other. Finally, in section 5.8 the game simulator of JUNTA is presented.

5.1 Scanner

This section presents our implementation of a scanner for our programming language.

Our scanner takes raw source code in the form of a program written in JUNTA as input and validates the lexical correctness of a JUNTA program. It does so by identifying tokens defined in JUNTA from the input. If an input is met which cannot be recognized as a valid token, the source code is not a valid program in JUNTA, hence an exception is thrown and scanning stops. The exception contains information about the line and offset at which the input was determined to be incorrect.

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 will be treated as an **ID** token. The name of the identifier will then be saved as a value belonging to the particular token.

Our scanner consists of two classes: **Scanner** and **Token**. **Token** contains an instance variable of the type **Type** which is an enumerate describing the kind of token.

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 following arguments: (**Type tokenType**, **String value**, **int** *line*, **int** *offset*). **value** is used in some cases for saving the lexeme the token was made from. For instance, the value of an **INT_LIT** (integer) or the name of an **ID** (identifier) must be kept. The *line* and *offset* represent where in the source code the lexeme of any token was found. This is essential if an error is found, since it is possible for the scanner to inform the programmer where in the source code an error exists.

When an input is to be analysed the scanner looks at the first symbol of the input, which is not a blank or white space character. Based on the first character met it transfers control to a particular method which is responsible for determining which kind of token that corresponds to the lexeme. This is actually an implementation of an explicitly controlled finite automaton[3]. The explicit control means that the transitions are handled "manually" by transferring control to appropriate methods rather than using a table driven automaton where lookups in a transition table causes a change of state.

In listing 5.1 you can see many methods named **is. . .()** for instance **isDigit()** and **isOperator()**. These are the functions which control is transferred to when the first character of a lexeme is a digit or an operator. If at any time the scanner has reached the end of the input stream, it returns a **EOF** so succeeding parts of the parser knows where the program ends. The method **isWhitespace()** is simply used to pop all white spaces between lexemes.

Note that a method which the control is transferred to will not always return the same type of token. For instance, the method **scanUppercase()** is responsible for determining whether the lexeme is a **type** or a **coordinate** because they are the only two tokens starting with an upper case character in JUNTA. This is depicted in table 5.1.

	Action and output	
First character	Responsible function	Token type
White space		*ignored*
End of file		EOF
Digit	scanNumeric()	LIT_INT
Uppercase	scanUppercase()	LIT_COORD, TYPE
"	scanString()	LIT_STRING
$	scanVar()	VAR
Operator	scanOperator()	OP_PLUS, OP_MINUS, OP_MULT, OP_DIV, OP_MODULO, OP_ASSIGN, . . .
Lowercase	scanKeyword()	KEY_THIS, KEY_SUPER, KEY_DEFINE, KEY_ABSTRACT, KEY_EXTENDS, . . .

Table 5.1: *The different groupings of characters, the method responsible for handling them and the possible returned token types.*

Two variables, **int offset** and **int line**, keep track of where in the source file the next input character is taken from. If an unexpected input symbol is met, a **SyntaxError** exception is thrown. An example could be a $ character followed by white space. $ is used as prefix for variables, and expects to be followed by an identifier, e.g. **$varName**. For more information on our error handling, see section 5.6.

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  }
```

When a lexically correct program is accepted by the scanner, a stream of tokens is created. These tokens are then fed to the parser, which analyses them and creates an abstract syntax tree of nodes based on the supplied tokens.

5.2 Parser

In this section we present Junta's handwritten parser. We have written a top-down recursive descent parser, which is within the class of LL(1)-parsers. The grammar for Junta is suited for this because it does not have left-recursive productions. In the end of the section we present our work with SableCC and the reason why we choose not to use this tool to make our scanner and parser.

5.2.1 Constructing the parser

The parser was very simple to implement, because it is structured exactly the same way as the grammar which can be found in appendix B. 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** which is the token name for a left bracket. If the grammar then expects a non-terminal, then the parser simply calls the method for that non-terminal, which may in terms have the effect of calling more non-terminals and expecting terminals (which are recognised as tokens) at some points, before continuing parsing the next part of the rule.

In listing 5.2, we give an example of how this structure looks like in our handwritten parser. The production rule for an if expression is presented in section 4.3.4.

The production for if expression says that every expression of this type must start with the combination of the two symbols spelling the word `"if"`. When the parser meets this word in an expression, it knows that it has to parse an if expression.

Listing 5.2: *How if expressions are parsed using top-down parsing in Java.*

```
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 }
```

5.2.2 Building an abstract syntax tree

In listing 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**). 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. If everything is syntactically correct the parser constructs a node for the AST for the given expression. The first child of the node is the boolean expression, and the next two siblings of that child are the expression branches of the if expression.

Terminal and non-terminals

Every grammar has a finite set of non-terminals and terminals. A production rule specifies a non-terminal as a sequence of non-terminals and terminals. We have defined tokens in the parser for every non-terminal in our grammar. The if expression has the token name **IF_EXPR**.

In the production for the if expression, we have three terminals: the `"if"`, `"then"`, and the `"else"`. These are all required 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.

5.2.3 Looking ahead in the input

We mentioned earlier that the parser is an LL(1) parser, which means that the parser only needs to look at the first token in the input stream before deciding what production rule to apply. We have shown the **lookAheadAtomic** method which determines if the next token is part of an atomic expression. The production for the atomic expression is presented in section 4.3.1.

An atomic expression can derive quite a few productions. This is why we have constructed a specific method to determine whether the next token is part of an atomic expression. This method is shown in listing 5.3.

Listing 5.3: *The lookAhead method to determine if the next expression is an atomic type.*

```
1  private boolean lookAheadAtomic() {
2    return lookAhead(Token.Type.LPAREN)
3        || lookAhead(Token.Type.VAR)
4        || lookAhead(Token.Type.LBRACKET)
5        || lookAhead(Token.Type.PATTERN_BEGIN)
6        || lookAhead(Token.Type.THIS)
7        || lookAhead(Token.Type.SUPER)
8        || lookAheadLiteral()
9        || lookAhead(Token.Type.TYPE)
10       || lookAhead(Token.Type.CONSTANT);
11 }
```

The method **lookAheadAtomic** makes use of two methods to figure out if the next token is part of an atomic expression. The first method is the **lookAhead** method that takes a token as argument and figures out if it equals the next token in the sequence of tokens. The second method is the **lookAheadLiteral** method which is similar to the method in listing 5.3 but instead of checking for atomic expressions it checks for literals. All these methods return true or false.

In listing 5.4 we show an example of how the **lookAhead** method is used in the parser. The example is taken from the **expression** method. The productions for expressions are presented in section 4.3. The production of an expression is reflected in the code of the parser. An example of this is given in listing 5.4.

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

```
1  private AstNode expression() throws SyntaxError {
2    if (lookAhead(Token.Type.LET)) {
3      return assignment();
4    }
5    else if (lookAhead(Token.Type.IF)) {
6      return ifExpression();
7    }
8    else if (lookAhead(Token.Type.LAMBDA_BEGIN)) {
9      return lambdaExpression();
10   }
11   {...}
12   else if (lookAheadAtomic() || lookAhead(Token.Type.OP_MINUS)) {
13     return loSequence();
14   }
15   {...}
16 }
```

The code presented in listing 5.4 is a small section of the **expression** method. We have removed code from the section which is not relevant for the example we are trying to give. The removed code is presented as {...}. An example is the code left out in the bottom which is just related to error handling.

An assignment begins with the reserved word `"let"` and the first **lookAhead** method peeks for that particular token to determine if the next production is an assignment. If the method returns true, then the next token is in fact the `"let"` word, and the parser enters a new method, namely the **assignment** method which parses the assignment and checks that the next tokens from the input complies with the production rule of an assignemnt. The same is done for the if expression, lambda expression, and operations which begins with the **loSequence()** (logical operators).

The operation production is a bit different, because it needs two **lookAhead** methods to determine if the next production is an operation (notice that this does not mean that it looks two tokens ahead). An operation can begin with either an atomic value or a minus operator. So the code uses a **lookAheadAtomic** and a regular **lookAhead** with the specific token as a parameter to check if the next production is an operation.

The methods return nodes that may have children nodes. These nodes are all connected to form a tree with a single root. This tree is called an AST and consists of nodes for all non-terminals and leaves for all terminals, which is the tokens generated by the scanner. When the AST has been constructed, it can be examined to verify that the input program has been parsed correct. The class **PrettyPrinter** has been implemented to output the structure and contents of an AST, making it easier to verify that the parser parses correctly.

5.2.4 SableCC

We have also implemented a scanner and parser using a compiler/interpreter generator, also known as a compiler compiler, namely SableCC[30]. As described in section 2.4.4, it is an automated scanner and LALR(1) parser generator written in Java, with support for making compilers and interpreters. We have implemented an early version of JUNTA in SableCC to evaluate the capabilities of such a tool.

Choice of SableCC

We chose SableCC instead of various other popular tools such as ANTLR[23] and JavaCC[21]. Though ANTLR and JavaCC are far more well-documented than SableCC, we still choose to work with this tool because its parser generates a LALR(1) parser which is generally a stronger parser than the LL(1) parser we wrote by hand or the LL(k) parsers generated by ANTLR and JavaCC. One of the major advantages is the LALR-parser's ability to handle left-recursion, which obviates left-factorisation.

SableCC outputs an abstract syntax node type for each alternation in every rule in the grammar specifications file, which is merely a simple text file with EBNF-like syntax. It's then possible to iterate over these nodes by extending the visitor pattern that SableCC also supplies, generating code or directly interpreting a syntactically correct program. This is all done in classes separate from the grammar specifications, which is also desirable and different from ANTLR and JavaCC, where action code is injected directly in the grammar specification. This is all done in Java, which is also desirable, since it would work well with the rest of the project, written in Java.

Experience with SableCC

Our experience with the tool has been rather cumbersome, in that it took quite a while to read the documentation before and while writing the specifications, as it simply isn't just copy/pasting EBNF grammar into a file. An example of the if-expression rule is seen below:

Listing 5.5: *Part of the grammar specifications file of SableCC, with focus on if expressions.*

```
1 Tokens
2   else          = 'else';
3   then          = 'then';
4   if            = 'if';
5
6 Productions
7   expression   = {elopexp} element operator expression
8                | {assign} assignment
```

```
 9                | {if} if_expr
10                | {lambda} lambda_expr
11                | {el} element list?
12                | {not} not expression;
13   if_expr      = if [left]:expression then [mid]:expression else [right]:expression;
```

This example brings out the strengths of SableCC, as it looks very similar to the EBNF for expressions and if expressions seen in section 4.3, with a few additions. As long as you know the special syntax and how helpers, tokens, and productions work, it is possible to create scanners and parsers very quickly. This was not our case, as no one had experience in using a compiler-compiler. The fact that there's a clear and clean separation between automated, generated code and user code makes the grammar and compilation/interpretation parts easier to maintain. When adding new features to the language, you simply have to update the specifications file and generate a new scanner/parser combination. Whereas when adding new features to the language while using a handwritten scanner and parser many lines of code needs to be change in order to implement the new feature.

Even though SableCC looks like a prime candidate to continue interpretation with, we chose not to use the tool. This is because it took an unreasonable amount of time to figure out how to precisely define the grammar correctly in SableCC due to their poor documentation. Also, it offers less control and customisability, compared to writing our own from scratch. As an example, the tool offers an application-specific interface to tree walking the AST nodes with the visitor pattern, requiring knowledge of how SableCC implements it. SableCC also generates around 17,000 lines of Java code, even for our simple grammar, which seems superfluous compared to our handwritten scanner and parser, consisting of "only" around 1500 lines of code.

Discussion of using a compiler compiler

We chose not to continue using SableCC on our updated grammar, due to the weight of cons against pros, and the fact that the time spent working on implementing SableCC was also spent making the handwritten scanner and parser and making them work exactly the way we want them to. It might not be as quick to make changes to our language with this solution, but the time spent on modifying and adding features to our language with the use of a handwritten scanner and parser is not wasted time, as it was spent learning, which gives us better understanding of their underlying functionality.

5.3 Abstract node types

This section illustrates how we have chosen to implement the grammar described throughout chapter 4 into abstract syntax trees (ASTs). We have split every program part into their own subsections with an abstract node type (ANT) that presents the program parts' construction when they are parsed into an AST. Each node type corresponds to a production in the grammar. These ANTs can then be combined to form the AST when some piece of code has been parsed.

Firstly, the section begins with a description of how an AST differs from a parse tree to make this clear to the reader. Then we present a few examples of our abstract node types. We do not present all the ANTs in this section because they are very similar to each other. The rest of the ANTs are in appendix A.

5.3.1 Difference between an abstract syntax tree and a parse tree

When a piece of code is parsed by a parser that understands the specific programming language, the output of the parser will be an AST, which consists of the abstract node types of each program part. The difference between an AST and a parse tree is that a parse tree contains every detail of the input code. A parse tree includes for instance the parentheses and keywords where an AST does not contain anything other than the abstract node types[28].

5.3.2 The abstract node type for program

Every program written in Junta begins with an ANT which we call "program" that consists of either zero, one, or more definitions. The production for this rule is presented in section 4.4.

It is from this production each and every program is derived from. The ANT for this production is illustrated in figure 5.1.

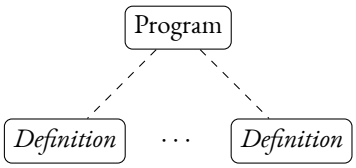


Figure 5.1: *The abstract syntax tree for the program node.*

Figure 5.1 consists of one root which is called "Program" and this root can have zero, one, or more children, called "Definition". The children are optional because the production says that a program can consist of either zero, one, or more of these definitions. We illustrate this choice by making the connecting edges dashed from the parent node. Between these two child nodes there are three dots (· · ·) which illustrate that it is possible to have more of these nodes following each other.

This means that a program is legal if it does not contain anything at all.

5.3.3 Abstract node type for operations with precedence and negation

In this section we introduce the five different groupings of operations such as logical operators, equality operators, etc., and finally we present the production called negation. We illustrate the five groupings of operators but we have omitted the negation node, because it is merely a root with one child. The grammar for this expression is presented in section 4.3.7.

The sequences are intentionally placed in specific orders to ensure the correct precedence for these operators.

Since the productions look a lot like each other we will only illustrate an ANT which shows how we have implemented the different operations. Figure 5.2 illustrates that the ANT for each production, with the left-hand side (LHS) of the production as the root, will only be constructed if there is at least two of the right-hand sides (RHS) with one operator between them. This means that the sequence should for instance look like the following:

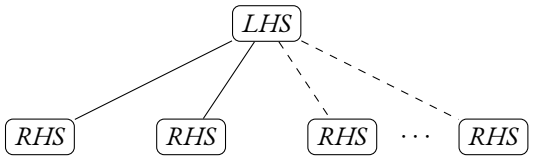


Figure 5.2: *The abstract AST for the different operation nodes.*

$$lo\_sequence \quad \rightarrow \quad eq\_sequence \texttt{ "and" } eq\_sequence$$

So, in the above example the RHS is $eq\_sequence$ and the operator is `"and"`. In figure 5.2 we do not show the operators, which can make it a bit cryptic to look at and understand. If there is only one RHS, then the node is not constructed but the next production will be evaluated. Otherwise the AST would end up with many single-child nodes. This bad implementation is illustrated in figure 5.3.

Figure 5.3 illustrates very clearly that the AST quickly would end up with many single child nodes. If we do not expect two RHS then we will end up with a long list, which is not necessary and it will just make it less efficient to traverse the AST. With our implementation we will have a more efficient and more compact AST.

The grammar specified earlier in this section presented that the last operation ($md\_sequence$) consists of negations and the choice to add an operator between negations. The negation production can be an element or begin with the `"-"` symbol followed by another negation. The production for a negation is presented in section 4.3.7.

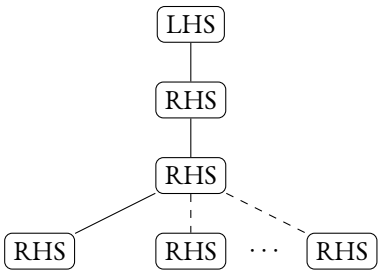


Figure 5.3: *An example of why we need at least two RHS before we construct a node.*

5.3.4 Abstract node type for let expressions

The production for the let expression is presented in section 4.3.3. Let expressions were named "assignment" in the Java implementation of the parser because the expression is a bit similar to an assignment. When we refer to an assignment expression we mean a let expression.

The production specifies that any let expression must begin with the keyword `"let"` and end with the keyword `"in"` followed by an expression. In between these beginning and ending keywords, the production consists of at least one sequence of a variable followed by an assignment operator followed by an expression. The production specifies that it is possible to have zero, one, or more of these variable expression pairs (comma separated) following the first pair.

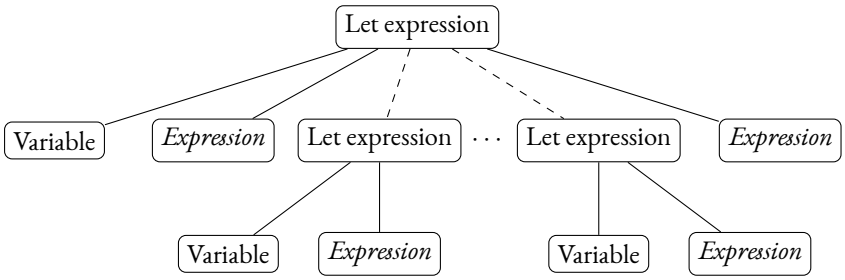


Figure 5.4: *The abstract syntax tree for the let expression node.*

Figure 5.4 illustrates this with an ANT that omits the keywords, commas, and the assignment operators, which can make it rather complex to look at. The figure actually states that a let expression consists of variables and expressions where an expression can be many different things, including another let expression. So, this means that it is possible to have let expressions nested within each other.

We have chosen to implement the optional additional variable declarations as new let expression nodes which each have two nodes that are not optional. These nested let expressions are connected to the parent with dashed edges which mean that they are optional and it is possible to have zero, one, or more of these following each other.

The rest of the ANTs are very uniform and they can be found in appendix A.

5.4 Contextual constraints

The **ScopeChecker** is the class responsible for enforcing some of the static semantic rules of JUNTA (described in chapter 4) at compile time. This section aims to explain how these static semantic checks are performed by the **ScopeChecker** in simple sequential steps. Any error detected by the **ScopeChecker** will cause a **ScopeError** exception to be thrown. This error contains helpful information about the type of error and where in the input program the error is located. The checking of static semantics as well as the interpretation of the code both use the visitor pattern which is a commonly used approach for both purposes.

After the AST has been created, the visitor pattern allows us to traverse the AST and execute encapsulated pieces of code for each specific type of **AstNode**.

5.4.1 TypeVisitor

The first visitor used is the **TypeVisitor**. This visitor traverses the AST, finds all type definitions in the input program and for each type definition an object of class **TypeSymbolInfo** is instantiated. After running the **TypeVisitor**, each type definition in the input program has an associated object of class **TypeSymbolInfo** which makes it easy to get information about any declared type in the input program by accessing the following members contained in the **TypeSymbolInfo** object:

- name (**String**): The type's name
- parentName (**String**): The name of its super type (null if not a derived type)
- args (**Integer**): The number of arguments in the type's constructor
- parentArgs (**Integer**): The number of arguments given in the call to its parent constructor
- data (**List of Data**): Each data defined in the type body has a corresponding **Data** object describing its name and position in the input program
  - The input program position is stored as a line and an offset and makes it possible to produce error messages with information about where in the input program an error was found
- members (**List of Member**): Each constant and function defined in the type body has a corresponding **Member** object describing its name, argument count, an abstract flag, a **TypeSymbolInfo** reference to the type defining the **Member** and a pointer to the line and offset in the code
- node (**AstNode**): A reference to the TYPE_DEF-**AstNode** which defines the type
  - This reference is used to get the input program position where the type was defined (for generating useful error messages), but is also used for marking abstract type definitions (described in detail in section 5.4.4)
- parent (**TypeSymbolInfo**): This will contain an object reference to its parent type if it has one
  - This is however a null pointer until running the **TypeParentRefMaker** described in section 5.4.2

All the **TypeSymbolInfo** objects are kept in an object of class **TypeTable**. The **TypeTable** class is a layer of abstraction which provides easy and fast access information about the types contained in the input program. The underlying implementation is a hashmap from the type's name as a **String** to its object reference, which provides a quick way to map a type's name to the **TypeSymbolInfo** object that represents the type. One use of this mapping is described in section 5.4.2. Another feature of **TypeTable** is a convenient way to iterate over all the **TypeSymbolInfo**s. This is for instance used to topologically sort the types, which are described in section 5.4.3. For convenience, we say that a type is added to a type table which means that a **TypeSymbolInfo** object representing the type is added to the **TypeTable** object representing the type table.

When a type is added to the type table, it is checked that no other types with the same name exist.

5.4.2 TypeParentRefMaker

The **TypeSymbolInfo** objects only contain the name of their super type as a **String** or a null value if there is no super type. By making a lookup in the **TypeTable** on the parent name, the real object references can be found and stored for faster and more convenient parent lookups which is used greatly by the visitor described in section 5.4.6.

5.4.3 TypeMemberPropagator

Some of the later checks that will be performed requires us to determine whether or not a type member (constant or function) with a specific name is visible in a given type. This requires searching in the given type and recursively in all super types for the member. When this kind of lookup is done many times on the same member, the traversal of the same long chains of parent references become inefficient which results in clumsy code. To simplify and speed up this process, the **TypeMemberPropagator** ensures that all members of a type A are also present in a type `C`, if `A` is a super type of `C`. With this approach, checking if a member is visible in type `C`, only requires looking in `C` instead of following the chain parent types.

This propagation of members is done by first topologically sort the **TypeSymbolInfo** objects, such that when iterating over the type table, any type yielded will always appear before all of its subtypes. This makes the afterwards propagation

of members possible in linear time by iterating over the topologically sorted types. If a type `C` is met, the members in its parent type `B` are put in `C` as well. If `B` has a parent `A`, we know that `B` has already the members from `A` due to the topological sorting.

The topological sorting is done using the algorithm that goes by the same name, from the book Introduction to Algorithms[24, p. 612] working on a graph $G = (V, E)$. Each type represents a vertex v. An edge e exists from v_1 to v_2 if the type v_2 is a parent of the type v_1. However this results in a topological order where a type always appears before its super types. This problem is quickly solved simply by reversing the list.

Given the graph in figure 5.5, the following sequences are examples of correct and wrong topologically sorted orders after the list has been reversed:

$$\begin{aligned} Correct &: \texttt{a, d, b, e, c, f} \\ Correct &: \texttt{a, b, e, c, f, d} \\ Wrong &: \texttt{a, b, c, f, e, d} \\ Wrong &: \texttt{b, c, e, a, f, d} \end{aligned}$$

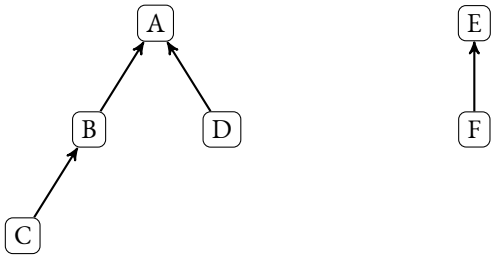


Figure 5.5: *Example of topologically sorted types. An edge goes from type X to type Y if X is a subtype of Y.*

Another great advantage from topological sorting is the fact that it reveals cycles in the graph. A cycle in the graph means an extend cycle between types exists, e.g: A extends B, B extends C and C extends A, which is not accepted.

5.4.4 AbstractTypeMarker

The interpreter needs to know whether or not a given type contains any abstract members. Such a type is an abstract type and should not be allowed to be instantiated. Marking these abstract types is now smooth. Due to the propagated members it can just be checked whether or not any abstract members are present in the given type. This check is done by the **AbstractTypeMarker**. Any type described by a **TypeSymbolInfo** has a reference to the **AstNode** it was defined from. If the type is found to be an abstract type, the type of the **AstNode** is changed from TYPE_DEF to ABSTRACT_TYPE_DEF. This is the only way to make information visible to the interpreter since the **Interpreter** does not use the same **TypeTable** class used by the **ScopeChecker**.

5.4.5 TypeSuperCallChecker

This checker ensures that any type that extends another type provides the right amount of arguments when calling the parent's constructor. A constructor can have x arguments and may or may not contain a variable amount of additional arguments. Consider the type constructor `Type A[$var1, $var2, ...$varargs]`. When calling the constructor from another type, e.g. `Type B[] extends A[5, 2, 7, 4]`, it must be checked that the type B provides *at least* the number of arguments in A's constructor (not counting the variable amount of extra arguments). If A does not have a variable amount of additional arguments, the argument count must match exactly. The implemented code for doing this check can be seen in listing 5.6.

Listing 5.6: *How the TypeSuperCallChecker is implemented.*

```
1 public class TypeSuperCallChecker {
```

```
2   public void check(TypeTable tt) throws ScopeError{
3     for (TypeSymbolInfo tsi : tt){
4       if (tsi.parent != null){
5         if (tsi.parent.varArgs == true){
6       //if parent type has x args and varargs, the subtype must provide at least x args in constructor call
7           if (tsi.parentCallArgs < tsi.parent.args)
8             throw new ScopeError("Number of arguments does not match for call to parent type " + tsi.parent.
                    name + " in type " + tsi.name + ". Expected a minimum of " + tsi.parent.args + ", received "
                    + tsi.parentCallArgs+".", tsi.line, tsi.offset);
9         }
10        else if (tsi.parentCallArgs != tsi.parent.args){
11          //if parent type has no varargs, the arg count must match exactly
12          throw new ScopeError("Number of arguments does not match for call to parent type " + tsi.parent.name
                  + " in type " + tsi.name + ". Expected " + tsi.parent.args + ", received " + tsi.
                 parentCallArgs+".", tsi.line, tsi.offset);
13        }
14      }
15    }
16  }
17 }
```

5.4.6 UsesAreDeclaredVisitor

This visitor ensures that any use of a variable, constant, function, data member, or type can be bound to a declaration. The visitor uses a variable (**TypeSymbolInfo currentType**) which updates upon visiting a TYPE_DEF or an ABSTRACT_TYPE_DEF **AstNode**, to keep track of which type it is currently visiting inside. If the visitor is not traversing inside a type (**TypeSymbolInfo currentType**) references a special type called **globalType**, which is used only to contain the standard- and game environment as well as the global constants and functions declared in a JUNTA game.

It is important to realise that **globalType** is not a super type of all other types, it is a stand alone type that no type can derive from. Its name contains an invalid character for a type name to ensure that no type can derive from it. This becomes handy when checking if constants and functions used can be bound to a declaration.

Constants and functions

When a constant or a function is referenced it is necessary to know two things about the context in which it was referenced:

1. In what type did the reference occur?
2. Is the reference a member access?

The first thing is easy to check since we have the **currentType** variable. This variable may however point to the global type. The structure of the AST makes it easy to determine if it was a member access, since we would have been visiting a MEMBER_ACCESS **AstNode** prior to the referenced constant or function. In the expression: `A[].B.C[2,3]`, both B and C are member accesses, but A is not. Given this information, a different check can be done regarding to the context of the reference:

1. Type was global and a member access
   - Must be visible in at least one type
2. Type was global but not a member access
   - Must be visible in the global scope
3. Type was A and a member access
   - If prefixed by this, it must be visible in A or any super type of A
   - If prefixed by super, it must be visible in any super type of A
   - If prefixed by a variable name, it must be visible in at least one type

4. Type was A but not a member access: Must be visible in A, a super type of A or global scope

One may wonder why an accessed member is accepted if the accessed member is visible in at least one type. Consider the member access `randomType.B`. Here it is unknown in what type we shall look for the member B. The constant `randomType` could literally return a random type, or the type returned could be determined by an arbitrary complex algorithm. Therefore, we can only enforce the rule that the member `B` must exist in at least one type.

Variables

For any variable, a declaration must always exist before it is used. A variable can only be declared in four ways:

1. As a type constructor
2. As a formal parameter in a function declaration
3. In a lambda expression
4. In a let expression

In all cases the Junta semantics require that a new scope is opened, in which the declared variable is known while the body of the expression is executed. When the scope closes the declared variables are removed. The body of an expression can also contain new variable declarations, e.g. a let expression in the body of a let expression.

The scope checker uses a **SymbolTable** class which is basically a symbol table with a list of variable names and a reference to a parent symbol table. The reference to the parent symbol table is exactly how the scopes inside other scopes are implemented.

```
type A[$a, $b]
{
  ...
}

define func[$a, $b] = ...

#[$a, $b] => ...

let $a = 1, $b = 2 in ...
```
(5.1)

Notice how the four code samples above all result in the same scope checking routine, which can be seen in figure 5.6. First, a new symbol table is instantiated in which the variables `$a` and `$b` are put in. The symbol table's parent reference is updated so it points to the current symbol table, which is referred by **SymbolTable currentST**. Next, the current symbol table is updated to the newly created symbol table, and the body (the triple dots) are executed. Lastly, the current scope is closed, which updates the current symbol table reference to point to the parent symbol table of the current symbol table. Notice that the symbol tables maintain a stack-like structure, where opening a scope pushes a symbol table on the stack and closing a scope pops one. The variable **SymbolTable currentST** points to the element on top of the stack.

When a variable is used, it is checked that the variable exists in any of the symbol tables by first looking in the current symbol table and recursively following the parent reference until a null reference is found. If a variable declaration with the same name as the used variable cannot be found, an error is generated.

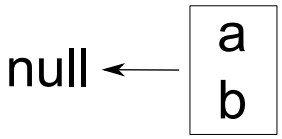


Figure 5.6: *Four different expressions that all result in the scope action depicted.*

It is important to realise the reason for maintaining the stack-like structure of symbol tables. It might seem like a single symbol table would be enough and that all variable declarations could just be put in there. This is indeed wrong, since the scope checker must also check for double declarations. A double declaration exists if a symbol table contains the same variable twice. Notice how figure 5.7 contains two symbol tables, each containing a declaration of $a.

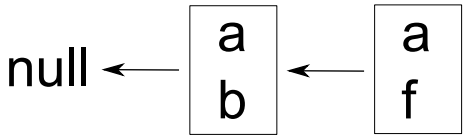


Figure 5.7: *The variable `$a` declared in two different scopes.*

This is completely valid and is caused by the following code sample. If only a single symbol table was used, an incorrect double declaration would be detected. See section 4.3.3 for more details about hiding.

```
let $a = 1, $b = 2 in (let $a = 3, $f = 4 in $a * $f + $b) + $a
```
(5.2)

Data members

When visiting a type body, a new scope is opened and the data members of **TypeSymbolInfo currentType** are immediately inserted into that scope. The children of the type body are then visited and the scope is closed. This ensures that the data members of a type can be used anywhere in the type body but in that type body only. When exiting that type body and closing the scope, the symbol table containing the data members are no longer visible.

5.4.7 Summary of scope checking

Many different static semantic checks are implemented in the scope checker. Though many other checks could have been included as well, the scope of the static semantics has been limited due to a few constraints. First of all, there is a deadline for this project, and with an almost endless set of semantic checks one can keep developing these checks. Furthermore, with new techniques being discovered once in a while, a compiler or interpreter can simply not include them all. A big set of the checks not included in Junta requires type checking, which is cumbersome in a dynamic programming language. However, it is generally possible to use type inference to find at least some of the types and errors associated with the use of them. It is important to realise that everything cannot always be inferred, for instance an algorithm could be so complex that it would need to be executed to determine all possible outcomes. Running the algorithm is not possible since you cannot know if the algorithm will ever halt[14, p. 173].

5.5 Interpreter

In this section we will look at the inner workings of the interpreter and see some examples of the different evaluation methods in use and how they're created. We introduce how we have implemented scopes and their symbol tables in section 5.5.1. Afterwards, we introduce our implementation of values and operators in section 5.5.2. Furthermore, we introduce our implementation of patterns in section 5.5.3. These patterns play a central role in most programs written in Junta.

The final step is the interpretation of the AST generated by the parser (and possibly modified by the scope checker). Here choices are made based on the different node types in the tree for a given program. This interpreter class is, like the scope checker, also implemented with the visitor pattern, visiting all the nodes and taking appropriate actions depending on what type of node is visited. The difference here is that values, types, and other language constructs are created and evaluated directly, propagating the results up the AST, and that every type of node is visited, as each node is significant in that they carry important information about the written program.

The implementation of the interpreter consists of 40 visitor methods, excluding private helper methods, each providing a specific behaviour depending on the type of node visited, possibly calling other visit methods on the node to evaluate to get a **Value**.

5.5.1 Symbol table and scopes

Symbol tables in the interpreter are used to add and get constants, variables, types, and to push and pop scopes. The interpreter keeps a single, global instance of a root symbol table which it calls methods upon when needed. This symbol table is unaffiliated with the scope checker's symbol table, which is constructed differently, and hence does not share any information with it at all.

Listing 5.7: *Code taken from the interpreter to show how the symbol table is used when visiting let-expressions.*

```
1  @Override
2  protected Value visitAssignment(AstNode node) throws StandardError {
3    symbolTable.openScope();
4    String var = node.get(0).value;
5    Value val = visit(node.get(1));
6    symbolTable.addVariable(var, val);
7    for (int i = 2; i < node.size() -1; i++) {
8      AstNode assign = node.get(i);
9      var = assign.get(0).value;
10     val = visit(assign.get(1));
11     symbolTable.addVariable(var, val);
12   }
13   Value ret = visit(node.get(node.size() - 1));
14   symbolTable.closeScope();
15   return ret;
16 }
```

Listing 5.7 shows how let expressions are evaluated when the abstract node type shown in section 4.3.3 is visited. A new scope is pushed onto the current stack of scopes, where after the variables are pushed into the newly opened scope. Because let expressions, like all JUNTA's expressions, return a **Value**, this value is retrieved by visiting the body of the assignment. When visiting the body, the variables of the assignment and their values are known, because they were put in the active symbol table immediately before visiting the assignment body. After having visited the assignment body, the symbol table is popped off the stack of symbol tables by calling **symbolTable.closeScope()**.

5.5.2 Values and their operators

Each of the base values offered by JUNTA is represented internally by a class with the name of the base value appended by **Value**. These classes are all subclasses of a general class **Value**, that offers the subclasses an interface to implement various different operations, such as comparison, addition, calling, and so on, throwing an error if trying to use the operator between incompatible values or if the operation yields an invalid result.

In the following part of this section, a few important values and their features are highlighted. The most basic values such as integers, strings, directions, coordinates, and lists principally support the same operations, and are therefore relatively uninteresting to discuss compared to the more complex values, such as types and functions. For completeness, the implementation of coordinates, types, and functions are discussed below.

Most properties (variables) of values are declared as Java **final**, due to the fact that JUNTA is functional and cannot modify the variables contained in a value. Instead, a new value is returned with the modifications of the original object. This may over time leave many dead objects on the heap for Java's garbage collection to deallocate.

Coordinates

Coordinates are represented internally by a tuple (x, y) specifying a coordinate on the game board. The maximum board size is limited by an x and y, which are both 32-bit signed two's complement Java integers [1]. If we wished to represent larger boards, we could simply use a built-in larger natural number representation, or create a custom representation.

As specified in section 4.6.1, coordinates consist of a horizontal axis represented by a letter, or multiple letters, x, and a numeral y, representing the vertical squares on a grid-shaped board. When displaying a coordinate to the user, a simple method is called to convert the x coordinate to its alphabetical form.

[1] Which gives a maximum coordinate of $(2^{31} - 1, 2^{31} - 1)$. Much more than any realistically imaginable board game!

Coordinate values support equals comparison, done by value. It also allows addition with strings, directions, and lists which is described in section 4.3.7. Listing 5.8 shows the implementation of the subtraction operation on coordinates.

Listing 5.8: *How subtraction of other values on Coordinate is handled.*

```
1  /** {@inheritDoc}  */
2  @Override
3  public Value subtract(Value other) throws StandardError {
4    if(other.is(CoordValue.type())) {
5      CoordValue oCoord = (CoordValue)other.as(CoordValue.type());
6      return new DirValue(x - oCoord.getX(), y - oCoord.getY());
7    }
8    else if(other.is(DirValue.type())) {
9      DirValue oDir = (DirValue)other;
10     return new CoordValue(x - oDir.getX(), y - oDir.getY());
11   }
12   throw new TypeError("Cannot subtract a " + other + " from a coordinate");
13 }
```

The method shown in the listing is principally the same implementation for every mathematical operation across all the different above-mentioned subclasses of **Value**. Type checking is always done to see if the right-hand side (RHS) of the operation is compatible before taking the correct actions. The method **is** on all **Value** types, checks to see if the type is the specified type or any subtype extending the type in JUNTA. If the RHS is allowed, it is cast up to the appropriate type if it isn't used directly, hereafter the appropriate operation is performed between the two types.

TYPES

Type values are one of the more complex subclasses of **Value**, as they, when defined, can have formal parameters, a parent **Type**, and a complex body. These are all represented and stored internally in a class **Type**, instantiated when visiting a type definition node (defined in appendix A.4).

There are a handful of different ways to instantiate a **Type**, all depending on the contents of the type definition node visited. If a type is declared to have a parent, a **Value** of type **Type** is simply instantiated by the interpreter with the appropriate constructor, saving the nodes for the parent and the parents formal parameters. If the type declaration also has a body, then each declaration is added to the newly instantiated type by the interpreter. Finally, the type is added to the symbol table, so it can be called and instantiated at a later point in the program.

Types are implemented to allow instantiating the type, returning a **ObjectValue** subclass with the actual parameters bound to the formal parameters, scope bindings, and a reference to the interpreter so it can evaluate the methods in its body (if one exists) when called. This instantiation is done when the interpreter visits an actual type, where it looks up the type's name in the symbol table, returning it if found, or an error if not.

Abstract type values are an extension of **Type**, but only consisting of abstract members, and do not allow instantiation, throwing a **TypeError** if attempted.

Unfortunately, the methods that make up the interface to the **Type** class are too large to be shown in this section, therefore we refer to the **TypeValue** class in the source code for a quick overview. The body of types can consist of constants and functions, where the implementation of functions is described below.

FUNCTIONS

Functions are yet another subclass of the **Value** type, though only supporting the addition operation to store the function in a **ListValue** (letting functions be first-class citizens). Functions are represented internally as constants, and can therefore be defined in the global scope or within any subscope that may exist, each updating the symbol table's current scope's constant declarations.

Functions are created with the AST node expression body, which is stored and not evaluated until the function is called. Functions do not keep track of their own names, making it the interpreter's symbol table's job. This allows lambda expressions to also have the same type. Functions therefore merely incorporate formal parameters and an expression, the body of the function. Functions also allow the ability to have a variable list of formal parameters. If the function is

a lambda expression, then the current scope is passed to the **FunValue** when instantiating. This is because the lambda expression is evaluated when declared. On the other hand, if a function is declared with a name identifying it, the interpreter adds it to the symbol table's constant table (thereby also keeping track of scope), instantiating a **FunValue** with the AST node expression and AST node formal parameters allowing it to be accessed in the symbol table when called.

When a function is executed, it is called with a reference to the interpreter and the actual parameters passed to the function. The interpreter is needed to evaluate the body, and is used for access to the symbol table for scopes to add and look up variables and functions, and so forth. For optimisation, an error is thrown if the length of the actual parameter(s) does/do not match the length of the formal parameter(s), already short-circuiting the call if attempted. Else the function body is evaluated with the actual parameters in a new, temporary scope that only exists while the call is running. This is done by calling the interpreter's visit method on the root body node, resulting in a **Value** that the function returns. This scope has the previous scope as a parent, giving it complete access to all the existing, declared members outside the function's body.

An interesting feature of the interpreter in respect to functions is tail recursion detection and optimisation. Special action is taken during the call when this form of recursion is detected, converting it to an iterative loop instead. An example of a tail recursive function, a small method **recsum** that makes use of it, is seen below:

```
define recsum[$x, $runningTotal] =
  if $x == 0 then
    $runningTotal
  else
    recsum[$x - 1, $runningTotal + $x]

recsum[5, 0] // returns 5 + 4 + 3 + 2 + 1 = 15
```
(5.3)

This simple function adds the first N integers, where N is defined as the first parameter to the function. The tail recursion optimiser converts the recursive function into a loop if it detects a tail recursion construction, guaranteeing stack overflow prevention when using tail recursion in Junta. How tail recursion is detected by the function when it is called is shown by the pseudocode below:

Listing 5.9: *Detecting tail recursion during call of a function.*

```
1  if 'second time we see this function' then
2    return new CallValue(interpreter, this, actualParam)
3
4  body := 'the AST node body of this function'
5  openScope()
6
7  recursion := false
8  repeat
9    For i := 0 to 'formal parameter size' n do
10     currentScope.addVariable(formalParam.i, actualParam.i)
11   End
12
13   evaluatedBody := visit(body)
14
15   if evaluatedBody 'is an instance of CallValue' and evaluatedBody.getFunction() = this then
16     recursion := true
17     actualParameters := evaluatedBody.getParameters()
18   else
19     recursion := false
20 until recursion = false
21 closeScope()
22 return evaluatedBody
```

Here it is important to note that an internal **Value** class **CallValue** is used to keep track of the returned body after being evaluated, making the check on line number 15 in listing 5.9 possible and true if it's the second or more time executing the body. This is done by having a boolean outside the call initially set as false, and being switched to true upon

executing the body. Since a recursive function calls its own body again, a check on the boolean is done to see if this is the case (line 1). If the boolean is set, a different, internal value **CallValue** is created with a reference to the **Interpreter** and current **FunValue** classes, and the function's actual parameters, all possibly being effected by the function.

5.5.3 Pattern matching

Patterns in Junta are a very important feature, being used in pretty much every one of our test programs to evaluate winning conditions, find squares or actions that have a specific property, and various other constructs.

Failed implementation with finite automaton

We looked at other languages in an attempt to find an existing solution to a similar problem and quickly found that patterns should be very similar to regular expressions. Unfortunately, regular expressions actually do not share the same properties as our patterns, ending up giving us quite a headache when trying to use automaton to describe patterns. One of the primary troubles we had using automatons was the fact that automatons take an input stream and decide whether the input is to be accepted or rejected. For a pattern matching, an input stream can be seen as a path between two squares on the board. For a person who plays a Junta game, it is however not convenient that he has to specify the exact path he wants to move. Consider trying to move the knight from B1 to C3 in a game of chess. Such a move could be chosen by sequentially clicking the 4 squares B1, B2, B3, C3. This inconvenient approach would also lack the support of clicking a piece and have the squares it can move to highlighted. An automaton would be useful if we had decided to use backtracking to feed it with the set of all possible input streams. However, this approach is not serious, because of the infinite possibilities of input streams.

Actual implementation

The idea of a pattern match is explained in section 4.5. In the implementation, a pattern matching method takes a pattern, a game object, and a position as input and then returns either true or false, depending on how the pattern complies with the game objects (board, squares and pieces) relative to the input position.

To aid the understanding of how pattern matching is implemented, this small example is introduced before the full implementation, not using any specific board layout:

> When evaluating a pattern, a single branch is created containing the input coordinate. A branch can be seen as a thing that "digests" the pattern sequentially and moves around on the board accordingly. The branch is the thing verifying that the pattern complies with board set up. For instance, given the pattern `/n | w empty/` and the start position B2, one branch starts on B2, meets the `|`-operator, and then splits into two branches. One branch moves one square north, the other moves one square west. The two branches are then united to a single branch containing the set of coordinates {B3, A2} The next digested part of the pattern is the `empty` keyword. The `empty` keyword causes all coordinates in the current branch to be removed if the square on the board on that coordinate is not empty.

The pattern is now fully digested, and if the current branch still contains any coordinates, it means that it was somehow possible to make a route from the starting position that fully complied with the pattern, thus the pattern matching returns true. If the branch did not contain any coordinates after digesting the entire pattern, this means that it is in no way possible to choose a branch which complied with the pattern, then the pattern match returns false.

In our full implementation of how a pattern is matched, these notations are used:

- `B` : a branch containing a set of coordinates
- `p` : a pattern
- `c` : a coordinate
- `d` : a direction, e.g. n (*north*) or sw (*south-west*)
- `clone(p)` : returns an exact clone of `p`
- `evaluate(p, B)` : modifies the branch `B` in a way depending on the value of `p`
- `union(`B_1`, `B_2`)` : returns $B_1 = B_1 \cup B_2$

- `concat(p, c)` : returns $\underbrace{\texttt{p p} \ldots \texttt{p}}_{\text{c times}}$

The full implementation of a pattern match is explained in the following. When checking if a pattern `p` matches on a coordinate `c` given the game object `game`, this is what happens:

> A new branch `B` is created, containing the input coordinate `c`. `evaluate(p, B)` is called, and when the function returns, it is checked whether `B` contains any coordinates. If it does the pattern check returns true, and false otherwise.

The `evaluate(p, B)` function depends on the composition of `p` the pseudocode for its implementation can be seen by listing 5.10.

Listing 5.10: *Pseudocode of how a pattern is evaluated on a branch.*

```
evaluate(p, B)
{
  if p is pLeft '|' pRight then
    BClone = clone(B)
    evaluate(pLeft, B)
    evaluate(pRight, BClone)
    B = union(B, BClone}
  else if p is pBody'*' then
    B´ = B
    do
      BLast = clone(B)
      B´ = clone(B´)}
      evaluate(pBody, B´)
      union(B, B´)
    while
      BLast != B
  else if p is p'+' then
    evaluate(p p*, B)
  else if p is p'?' then
  else if p is pCONSTANT then
    evaluate(concat(p, CONSTANT), B)
  else if p is a direction d then
    For all coordinates c in B, c += d
  else if p is in {'friend', 'foe', 'empty', object, type} then
    For all coordinates c in B
      If the square at c does not comply with the check-value p then
        Remove c from B
  end if
}
```

Visualisation of pattern algorithm

The algorithm in listing 5.10 is used in figure 5.9 and figure 5.10 to visualise how the algorithm works. The patterns will be executed on the board in figure 5.8.

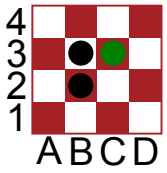


Figure 5.8: *A possible board set up.*

The execution of the algorithm trying to match the pattern `/s? w* empty n !empty/` on the square D3, can be seen in figure 5.9. The board set up used is that depicted in figure 4.4.

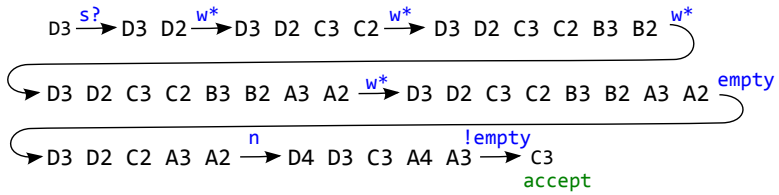


Figure 5.9: *Pattern matching of* `/s? w* empty n !empty/` *on the square D3.*

The execution of the algorithm trying to match the pattern `/n|w n|w !empty/` on the square C2 can be seen in figure 5.10. The board set up used is the one depicted in figure 4.4. The two branches created when meeting a `|`-operator are merged again shortly after. Notice how this increases efficiency, as the two red coordinates (B3) are merged into a single branch before trying to match the successive pattern values.

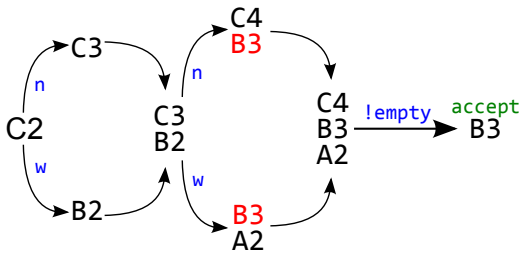


Figure 5.10: *Pattern matching of* `/n|w n|w !empty/` *on the square C2.*

5.5.4 Evaluation of interpreter

We have implemented a complete interpreter with semantics for every single node type, allowing us to explore and demonstrate every feature of JUNTA without many limits.

Under development, we have kept flexibility in mind, allowing us to easily add new base values and extend and maintain the feature set and semantics of our language, without having to change any existing code. If we wanted to add an addition operation between coordinates, it would only be a matter of adding a few lines of code to the coordinate value class.

It is obviously not the fastest implementation of an interpreter, but we don't see that as a hindrance, because speed is not what we're after.

There are a few optimisations that can be taken into account when writing an interpreter to make it run faster, such as detecting tail recursion. Since JUNTA has no loop-constructs, recursive functions are the only way to repeatedly run through some code.

Patterns are also a very important and central feature to our language. Though some small optimisations have been made, pattern matching requires a non-deterministic implementation, again hurting performance. A restriction with patterns is that they only work on grid-formed boards.

5.6 Error handling

In the different phases of the interpreter many different errors can occur. In the scanning phase for instance, a scanning error can occur and be thrown if the scanner detects a string, which doesn't correspond to one of JUNTA's specified lexemes. In the parsing phase a syntax error can occur if two or more tokens are not set up syntactically correct according to our grammar. In order to catch these errors we have implemented error handlers, which extends the built-in Java class **Exception** (**Java.lang.Exception**).

An error can cause serious flaws and is annoying for a programmer if no information of what caused it is provided. The function of the error handlers are to give the programmer of JUNTA games a better chance of figuring out what has been done wrong and make it easier to correct the errors by, for instance, referring to where in the source code an error has occurred and what caused it. We have created an error hierarchy in order to give an overview of our different error handlers. The hierarchy is illustrated in figure 5.11.

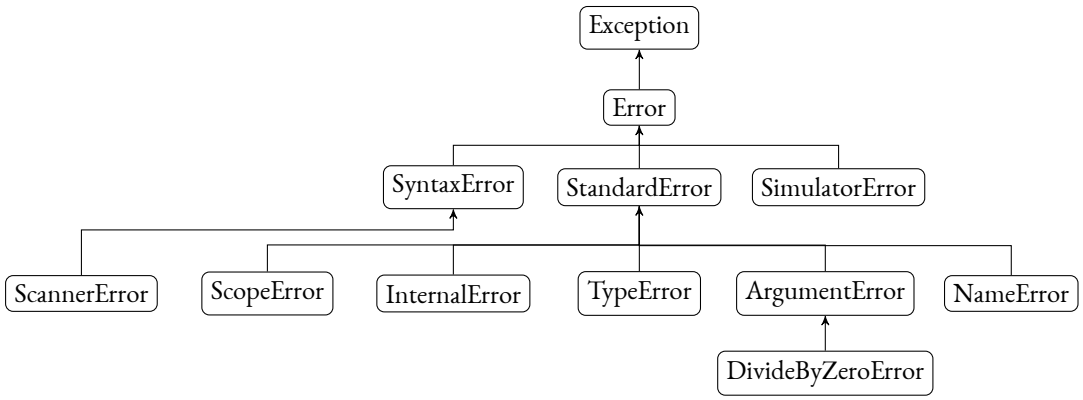


Figure 5.11: *An illustration of the error hierarchy as it is implemented in the interpreter.*

Exception is Java's standard error handling concept and is located at the top of the error hierarchy. Exception is a built-in Java class that extends the **Throwable** class which is the superclass of all errors and exceptions in Java. Only **Throwable** or subclasses of **Throwable** can be the argument type in a catch clause and only objects that are instances of this class or it's subclasses can be thrown by the Java throw statement[5].

Error extends the **Exception** class. **Error** is an abstract class which contains two abstract methods: **getColumn()** and **getLine()**, allowing subclasses to provide more detailed error messages.

SyntaxError extends the **Error** class and is used in the parser to handle syntactic errors. A syntactic error occurs if a token doesn't correspond to the currently expected token according to the grammar. **SyntaxError** outputs a message and information about the line and offset of the token that caused the error.

ScannerError extends the **SyntaxError** class and is used in the scanner to handle lexical errors. A lexical error occurs if the programmer makes a typo, e.g. writes "defin" instead of "define" or if it in other ways doesn't follow the regulations of JUNTA's lexemes e.g. writes the first letter in a type name in lower case letters. Like **SyntaxError**, **ScannerError** outputs a message and information about the line and offset of the token that cause the error.

StandardError extends the **Error** class. The **StandardError** class is used in the interpretation phase to handle errors with nodes in the abstract syntax tree.

ScopeError extends the **StandardError** class. It is thrown by the scope checker to notify the programmer of scope errors.

InternalError extends the **StandardError** class. It is an error that encapsulates Java's own exceptions, if they are thrown. It essentially represents an error in the implementation or environment in which it is running, and is not caused directly by the programmers code.

TypeError extends the **StandardError** class. It is thrown at runtime when a value is of the wrong type, for instance when using an operator such as the + operator, the operands must be applicable for that operator.

ArgumentError extends the **StarndardError** class. It is thrown at runtime if a function is supplied the wrong number of arguments.

NameError extends the **StandardError** class. It is thrown at runtime if a used constant, type or variable is not defined in the current scope.

DivideByZeroError extends the **ArgumentError** class. It is thrown when dividing by zero.

SimulatorError extends the **Error** class. It is thrown if the simulator encounters an error.

5.7 Game abstraction layer

We need an abstraction layer to act as the glue between the interpreter and simulator, allowing a simulator access to different elements in a written program. This has caused us to write an abstraction layer on top of the interpreter, offering interfaces such as our graphical simulator.

There are three different packages relating to the game abstraction layer. The first is the central class in the Game Abstraction Layer package, the second are wrappers in the Interpreter package, providing the actual implementation, and lastly is the game application programming interface in our Utilities package. Each is described in this section.

5.7.1 The main layer

The entire program package offered by JUNTA is encapsulated by the class **GameAbstractionLayer**. This small class is basically only defined by its constructor, seen below in listing 5.11:

Listing 5.11: *The game abstraction layer's constructor, initializing different constructs based off of an input of characters.*

```
1  private GameEnvironment env = new GameEnvironment();
2  private Interpreter interpreter = new Interpreter(env);
3
4  public GameAbstractionLayer(InputStream input) throws Error {
5    Scanner s = new Scanner(input);
6    LinkedList<Token> tokens = new LinkedList<Token>();
7    Token ts;
8    while ((ts = s.scan()).type != Token.Type.EOF) {
9      tokens.add(ts);
10   }
11   Parser p = new Parser();
12   AstNode ast = p.parse(tokens);
13   ScopeChecker scopeChecker = new ScopeChecker();
14   scopeChecker.visit(ast);
15   interpreter.visit(ast);
16 }
```

Here all the constructs are tied together: The handwritten scanner is instantiated with the input provided in the constructor, creating a stream of tokens fed to the parser, that then creates an abstract syntax tree traversed by the interpreter. The interpreter and its information is then used when getting the main game wrapper. The class is also complimented by the method **getGame**, called after instantiating the **GameAbstractionLayer**. This method returns a **GameWrapper** (described in the next subsection), containing all the needed information about the written game.

This class is what's called and used by the simulator when starting up, as the simulator needs access information about the game, which in turn builds on top of our scanner and parser. It's meant to be used by any interface that wishes to access and modify a game's state.

5.7.2 The application programming interface

The application programming interface (API) provided is simply a collection of interfaces used by the wrappers. This means that classes implementing these interfaces guarantees different methods are available. As an example, the **Square** interface is seen in listing 5.12 below:

Listing 5.12: *The Square interface, one among many such interfaces provided by* JUNTA.

```
1 public interface Square {
2   public int getX() throws StandardError;
3   public int getY() throws StandardError;
4   public Piece[] getPieces() throws StandardError;
5   public String getImage() throws StandardError;
6   public boolean isEmpty() throws StandardError;
7   public boolean isOccupied() throws StandardError;
8 }
```

Currently 12 such interfaces exist for different aspects of a game, such as getting information about the game itself, its players, board, action sequences, etc. These interfaces all build on top of the default types in the game environment described in section 4.6. As a rule, an interface exists for every default type defined in the standard environment. If we want to expand the standard environment, an interface and its wrapper would need to be added, considering the fact that it's one of the only ways 3rd party simulators can interact with JUNTA.

5.7.3 Wrappers

Wrappers provide a way of accessing the values created by the interpreter, allowing a simulator to use the properties of these values to, for example, display the game's title, supply information about the winning conditions, squares and pieces, actions and move history, and so on. The wrappers implement the interfaces in the API described above. The main wrapper returned from the method **getGame** provides access to all the other wrappers (implementing their respective interfaces) via methods in its body.

When instantiating **GameWrapper** in method **getGame** of class **GameAbstractionLayer**, an instance of the interpreter is passed to the method, which means full access to the symbol table and standard environment. All wrapper classes have the following signature:

Listing 5.13: *The signaure of all API wrapper classes.*

```
1  public class xxWrapper extends Wrapper implements yy { ... }
```

Where xx is one of the 12 wrappers and yy is the matching interface. The root class **Wrapper** houses a series of methods used by all the wrapper subclasses that retrieve different values in the type the specific wrapper abstracts over (as in the implemented interface).

This construction is the only way applications such as our simulator can interact with the programming language. A simple instantiation of **GameAbstractionLayer** provides everything needed for this to happen (granted it's a Java application).

5.8 Simulator

In this section, we begin by giving an overview of the set of classes in the simulator and their relations, in section 5.8.1. Afterwards, we present **Widget**s and their properties in section 5.8.2. We also explain the different actions that are available in section 5.8.3. Furthermore, we explain how the different **Widget**s communicate with each other in section 5.8.4. Lastly, we present how we have made the game interactive in section 5.8.5, and how everything is connected in section 5.8.6.

A simulator used to interact with the written board games has been written to play games and to test different features of our language. This is done by interacting with GAL, explained above. By using our API, switching out the simulator with another graphical interface is relatively easy. This is because interaction with the JUNTA game is done by interacting with the methods of the API.

The Simulator has been constructed to provide a visual interface to the API provided by the Game Abstraction Layer (GAL). The interface should give a visual representation of the board and pieces, similar to how the board game would look in real life. Furthermore, it should provide an easy way to interact with the game. This interaction should be sufficient enough to make it possible to play the game. The result is a Java application with a graphical user interface, which takes a JUNTA code file and makes it playable by the use of GAL.

5.8.1 Overview

The implementation is based around the class **Widget**, which simplifies the process of distributing drawings and handling user input. Some **Widget**s manage other **Widget**s, while others provide visual and interactive content. The different game **Widget**s provide visual and interactive interfaces to GAL.

To work with graphics and input, the game framework slick2d is used [20]. **SimulatedGame** is used to bind slick2d, **Widget**s, and GAL together to provide a complete system. Figure 5.12 shows all these components and their most important relationships.

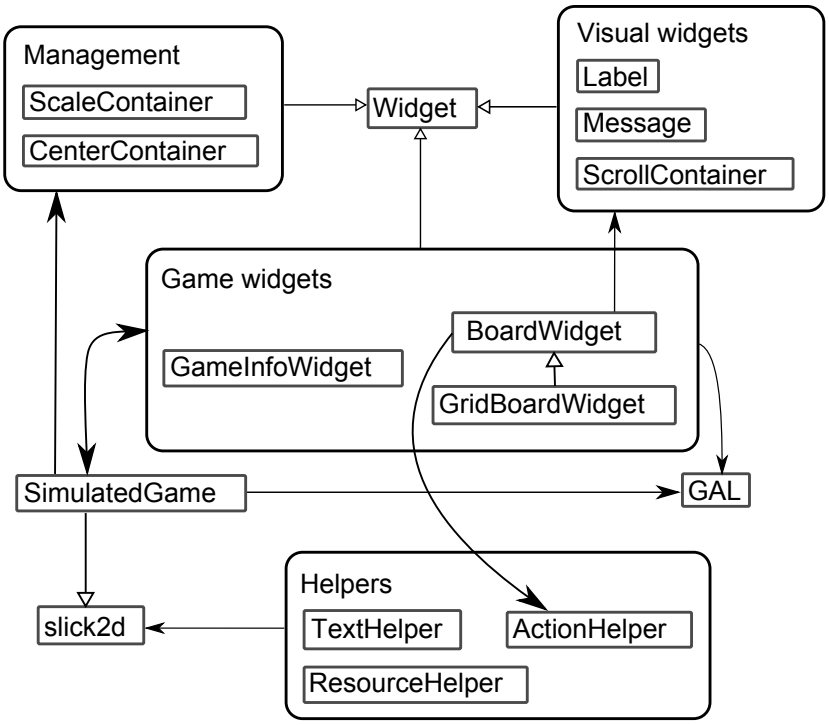


Figure 5.12: *Generalised overview of classes in the Simulator and their relations. Triangles means inheritance, while arrows means that one class makes use of the one it points to.*

5.8.2 Widget

Widget specifies an object with a size and a position, can be drawn, take mouse input, and send messages to other objects descending from **Widget**. The most important property of **Widget** however is that a **Widget** can contain several sub-**Widget**s which each can contain further sub-**Widget**s. We use this tree structure to control how drawing and mouse input is handled.

Placement

A **Widget** has a position specified with an (x, y) coordinate. Its position is relative to its parent, so if a **Widget** has position $(7, 10)$ and its parent has the position $(23, 50)$, its absolute position is $(30, 60)$.

Furthermore, a **Widget** has a size specified with a width and height, but it also contains an allowed range for each dimension. This allows us to specify that a **Widget** might be dynamic in size and can be adjusted if wanted.

Automatic placement and sizing

Instead of setting sizes and positions manually, we create container classes that manage the position and size of their contained **Widget**s. By using **Widget**s dynamic in size, we can create a layout that works independently of the window and board size.

ScaleContainer is such a container **Widget** that positions **Widget**s along an axis. For **Widget**s whose size is dynamic, the remaining available space is distributed evenly among them. An example is shown in figure 5.13. The top-level **Widget** is a **ScaleContainer**, set to position **Widget**s vertically. It does not effect its own size, only its sub-**Widget**s. The second **ScaleContainer** (containing two buttons, to be positioned horizontally) is thus resized by the first **ScaleContainer**. The ordering of the sub-**Widget**s determines the sequence they are positioned in in the **ScaleContainer**.

Creating a layout is now only a matter of building a hierarchy of **Widget**s, not deciding the exact position and size of each and every single **Widget**.

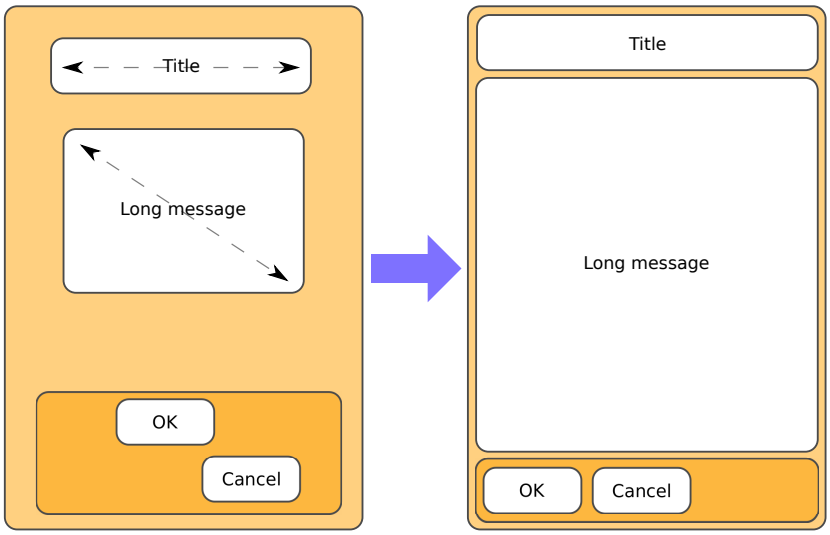


Figure 5.13: *How* ***ScaleContainers*** *(marked with colour) effect the positioning and resizing of sub-***Widgets***.*

5.8.3 Propagated actions

Drawing a **Widget** should not only draw the **Widget**, but also all its sub-**Widget**s and their sub-**Widget**s, and so on. To do this, **Widget** has two methods, **draw** and **handleDraw**. **handleDraw** needs to be overridden in sub-**Widget**s wanting to provide a custom drawing method. **draw** handles all the logic for drawing sub-**Widget**s, so the inherited class only needs to worry about itself. An overview of **draw** is given in listing 5.14.

Listing 5.14: *Pseudocode for the draw method.*

```
1  final public void draw(){
2    g.translate( getX(), getY() );
3
4    g.setClip( absX, absY, getWidth(), getHeight() );
5    handleDraw( g );
6    g.clearClip();
7
8    for( Widget o : widgets )
9      o.draw( g, absX + o.getX(), absY + o.getY() );
10
11   g.translate( -getX(), -getY() );
12 }
```

To further ease development, the coordinate system is translated, so **handleDraw** will be done using local coordinates instead of absolute coordinates. Furthermore, we apply clipping, so that any drawing outside the **Widget** will be clipped and not displayed. This way we can ensure that **Widget**s can't mess with other **Widget**s.

We enforce this by restricting the **draw** method with Java's final keyword so it can't be overwritten, and **handleDraw** is protected, so the calling class can't call **handleDraw** instead of **draw** by accident.

Mouse input

The same pattern is used for mouse input, but here we use it to determine which **Widget** is responsible for handling it. An overview is given in listing 5.15.

Listing 5.15: *Pseudocode for the mouseClicked method.*

```
1  final public boolean mouseClicked( int button, int x, int y ){
2    for( Widget o : widgets )
3      if( o.containsPoint( x, y ) )
4        if( o.mouseClicked( button, x - o.getX(), y - o.getY() ) )
```

```
5         return true;
6   return handleMouseClicked( button, x, y );
7 }
```

Like with drawing, we translate coordinates into local coordinates, however notice that the method returns a boolean, used to determine if the event was handled, and it will stop as soon as any callee returns true. A second difference is that in contrast to drawing, input is handled bottom up. The reasoning is that the lower we get in the hierarchy, the more specific the behaviour of each **Widget** is. Thus, we try to see if the more specific **Widget**s will handle the input and if not, less and less specific **Widget**s are tried.

When mouse buttons are pushed and released, it will only try **Widget**s containing the position at which the mouse is currently positioned at. For mouse dragging the situation is different, it will try any **Widget** which has initiated a drag, even if the mouse has moved outside it. If this was not the case, a scrollbar for example would only move if we kept the mouse exactly on top of it, which usually is tricky, as they are long and slim.

5.8.4 Communication between Widgets

Consider the case where a **Widget** represent a button. The user might click on it, but the button by itself is not interested in what this should signify. Thus, we need some way of notifying **Widget**s that some events have happened inside other **Widget**s. For this, the Observer design pattern is used in **Widget**.

5.8.5 Making games interactive

Two subclasses of **Widget** interacting with GAL are used to present the game to the user. They are **GameInfoWidget**s, which provide information like move history and the game description, and **BoardWidget**, which display an interactive board with pieces based on GAL. The class **GameInfoWidget** is not discusses further, as it simply calls getters from the GAL game object containing the information needed.

BoardWidget

For interaction, **BoardWidget** supports selecting **Action**s by the use of either Drag & Drop or Click & Select. While Drag & Drop only allows you to move a **Piece** from one **Square** to another, Click & Select will work on any two **Square**s, whether or not they contain any **Piece** or **Piece**s. It will go through all available **Action**s and find the ones related to the specific **Square**s. To help ease this process, usable **Square**s are hinted, as shown in figure 5.14.

Figure 5.14: *The Square at E2 was selected and the 4 pieces which can move there gets highlighted.*

In reality, we have two board **Widget**s: **BoardWidget** and **GridBoardWidget** (a specialisation of **BoardWidget**). While it is not necessary at this point to have this hierarchy, it is an attempt to generalise **GridBoard** so that a future addition with new **Board** types will be easier to implement. This is further described in section 7.1.4.

5.8.6 Binding everything together

The class **SimulatedGame** has the responsibility to connect slick2d with the **Widget** structure, and GAL with the game **Widget**s. **SimulatedGame** contains one **ScaleContainer**, which it resizes to fit the whole window, and tells it to adjust the sizes of its sub-**Widgets**. Secondly, it sends all mouse-events to this **Widget** and draws it whenever slick2d wants to be redrawn.

On construction of **SimulatedGame**, it reads the Junta code file and attempts to load it through GAL. It then creates the game **Widget**s, but it does not pass a reference to the game directly. This is because the game object changes each time an **Action** is applied, which requires us to update it in every game object each time the user for example moves a piece. Instead we pass a reference to this instance of **SimulatedGame** and the game **Widget**s must then access the game object through its accessor's methods directly, without caching it.

One final task of **SimulatedGame** is to handle any exceptions in GAL or the **Simulator** itself, and show them to the user without the application crashing. This is done by stopping the game displaying a pop-up window with the error's message printed in the window. The exact exception is also written to the Simulator's log, among other information.

5.9 An overview of the implementation of Junta

Now that the implementation of the different constructs that make up our implementation of Junta have been described, we wish to provide an overview of how they fit together, attempting to provide a concise overview. Figure 5.15 presents how these components actually are connected.

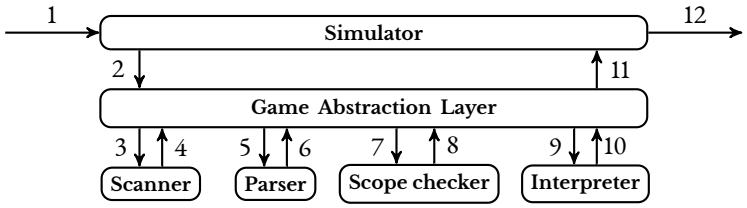


Figure 5.15: *A diagram showing the components of Junta and how they are connected.*

The numbers seen in the figure is the sequence of stages in which programs written in Junta will go through before they are playable. The first arrow (1) is when a user inputs a program and the last arrow (12) is when the program is ready to be played.

The following enumeration explains what each number the arrows have:

1. File location
2. Stream characters
3. Stream of characters
4. Stream of tokens
5. Stream of tokens
6. Abstract Syntax Tree
7. Abstract Syntax Tree
8. Abstract Syntax Tree
9. Symbol table and Abstract Syntax Tree
10. Symbol table
11. A game object
12. Fun with playable board games

The simulator begins by accepting a file location as input. Then the game abstraction layer (GAL) receives a stream of characters from this file at that location (the simulator opens the file and reads it, instantiating GAL with the input stream). The GAL forwards the stream of characters to the scanner, which scans the input and outputs a stream of tokens to the GAL. The GAL again forwards the stream of tokens to the parser, which parses the input and outputs

an abstract syntax tree (AST) to the GAL. The GAL once again forwards the AST to the scope checker, which checks the AST for errors and passes the checked (and possibly a modified) AST back to the GAL. One last time, the GAL forwards the AST to the interpreter (instantiated with the game environment), which visits every node and takes action depending on the node types, meanwhile building up its symbol table. The interpreter outputs the symbol table to GAL, and GAL sends an object with information about the game to the simulator, which now makes it possible to play the board game.

All this can of course be interrupted an error is thrown in one of the stages. It is up to the simulator to handle these errors. If no errors are encountered, then it is possible to see the game in the simulator and the visualised game can be played.

Chapter 6

Evaluation

This chapter aims to review the different aspects of the design and implementation of our programming language. Here we look at how easy writing games is and if they can compete with the same games written in other languages in section 6.1. Hereafter our simple unit testing framework implemented in Junta is described in section 6.2. Lastly, the interesting and important requirements listed in chapter 3 are assessed, to see how many we fulfill and which we haven't seen completed in the final implementation of Junta.

6.1 Writing games in Junta

We have written a handful of games in the final version Junta to gain a better practical understanding of writing in the language along with finding possible holes and features that may exist. The games vary from classic board games to small, unknown puzzle games invented by individual group members. All game screenshots in this section are taken from the simulator while running an implementation of a particular game.

This section introduces four of those games, namely Noughts and Crosses, Connect Four, Ice, and Kent's game. Some of these have their source code listed to show the simplicity of Junta.

6.1.1 Noughts and Crosses

The first game implemented in Junta is Noughts and Crosses. A picture of a possible board layout is shown below in figure 6.1. The source code for the entire game is presented in code sample (6.1):

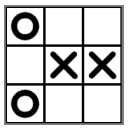


Figure 6.1: *An implementation of Noughts and Crosses in Junta, played in the simulator.*

6.1.2 Connect Four

The famous Connect Four game was written in approximately five minutes using the Noughts and Crosses implementation as a template, modifying board set up, win condition, etc. The game loaded in the simulator can be seen in figure 6.2. The source code is shown in code sample (6.2) underneath.

```
type NacGame[] extends Game["Noughts and Crosses"] {
  define players = [
    NacPlayer[Crosses, "Crosses"],
    NacPlayer[Noughts, "Noughts"]
  ]
  define initialBoard = GridBoard[3, 3]
}
type NacPlayer[$pieceType, $name] extends Player[$name] {
  define winCondition[$gameState] =
    $gameState.findSquares[/friend (n friend n)   |
                           (e friend e)   |
                           (nw friend nw) |
                           (ne friend ne ) friend/].size != 0
  define tieCondition[$gameState] = $gameState.board.isFull
  define actions[$gameState] = addActions[$pieceType[/this/],
                                          $gameState.board.emptySquares]
}
type Crosses[$owner] extends Piece[$owner]
type Noughts[$owner] extends Piece[$owner]
```

(6.1)

```
type ConnectFour[] extends Game["Connect Four"] {
  define players = [
    ConnectPlayer[Crosses, "Crosses"],
    ConnectPlayer[Noughts, "Noughts"]
  ]
  define initialBoard = GridBoard[8, 8]
    .setSquaresAt[Bottom[],
                  [A1, B1, C1, D1, E1, F1, G1, H1]]
}
type ConnectPlayer[$pieceType, $name] extends Player[$name] {
  define winCondition[$gameState] =
    $gameState.findSquares[/friend (n friend)3  (e friend)3  (nw friend)3
                           (ne friend)3/].size != 0
  define tieCondition[$gameState] = $gameState.board.isFull
  define actions[$gameState] = addActions[$pieceType[/this/],
                                          $gameState.findSquares[/empty s !empty/]]
}
type Crosses[$owner] extends Piece[$owner]
type Noughts[$owner] extends Piece[$owner]
type Bottom[] extends Square[] {
  define isEmpty = false
}
```

(6.2)

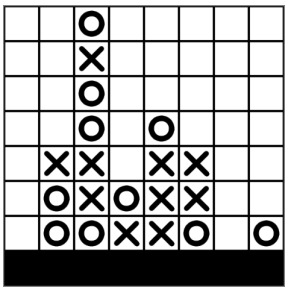


Figure 6.2: *An implementation of Connect Four JUNTA, played in the simulator.*

6.1.3 ICE

The Ice game is a single-player puzzle game partly invented by one of the group members. Partly here means that it is not based on any particular game, but the game mechanics is used in many other puzzle games. The goal is to move the player (the human-shaped piece) onto the green square. The player can move continuously in one of the directions north, east, west, or south until he meets a black wall. He cannot stop halfway on the path. A screenshot of the game loaded in the simulator can be seen in figure 6.3. The red numbers show the moves needed to solve the puzzle, they are not visible in the game.

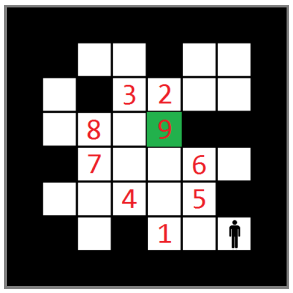


Figure 6.3: *A JUNTA implementation of a custom puzzle game, Ice, played in the simulator.*

6.1.4 KENT'S GAME

Kent's Game is another game invented by the group. Figure 6.4 shows the initial board set up. The goal is to swap the position of all red and blue pieces. A blue piece can move one square north or one square east if it lands on an empty square. It can however move two squares north or two squares east by jumping over a red piece, landing on an empty field. The moves of a red piece are identical to those of a blue piece, where moving is mirrored in the opposite direction. The game is hard to solve because pieces can never move back, and the narrow passage in the middle allows only one piece to pass at once. One of the group members liked the game idea and implemented it in JavaScript so he could post it on his website. He included a backtracking solver which verified that the game is solvable with multiple solutions after dozens of moves. The source code is not listed here, but is only about 35 lines. Below, in figure 6.4, the initial board setup in the simulator is shown.

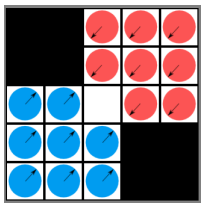


Figure 6.4: *A JUNTA implementation of a custom puzzle game, Kent's Game, played in the simulator.*

6.2 Unit testing

A built-in type in Junta makes it easy to unit test a game written in Junta. If a source code contains a type which extends the `TestCase` type, it means that any constant defined in the type must evaluate to true.

An example of a handful unit tests of Noughts and Crosses can be seen below:

```
type NacTest[] extends TestCase[] {
  data $state1 = NacGame[]
  define testTitle = $state1.title == "Noughts and Crosses"
  data $state2 = $state1.applyAction[$state1.players[0].actions[$state1][0]]
  define testEmptySquares = $state2.board.emptySquares.size == 8
}
```
(6.3)

The data member `$state1` contains a Noughts and Crosses game in its starting state. The first constant `testTitle` verifies that the game title really is set to "Noughts and Crosses". The data member `$state2` contains the game state after the first player has performed the first action in the list of possible actions. This first action will cause a piece to be placed on one of the squares, which means that $3 * 3 - 1 = 8$ squares must still be empty. This is verified by the test constant `testEmptySquares`.

This construct allows the programmer to define one or more test types for the game in question (in the same file, as Junta does not support multi-file imports). If one of the definitions within a type extending `TestCase` returns false or any other value that isn't equal to true, then the interpreter throws an error to notify the programmer of a failing test.

6.3 Requirements evaluation

In chapter 3, we formulated a series of requirements. The chapter featured requirements for the functionality and performance of Junta and also requirements for the final solution. In the following section we will evaluate the requirements. Those we find most interesting and essential for the project are evaluated more detailed than those which are less relevant.

Requirement 1a: One of the major requirements of the project is requirement 1a: "It must be possible to implement Chess, including the special rules of Chess". This however has not been possible, due to time pressure. The special rules of Chess, and especially *En Passant* and *Castling* moves proves to be difficult challenges, requiring a good amount of time to get implemented. We do though believe that Junta provides the features write a complex game such as chess.

Requirements 2-5: Are all accomplished.

Requirement 6: Are all accomplished. Requirement 6j: "Check if the current move about to be made for a piece is the first move made by that piece" is not directly implemented, but can be accomplished by for example using a counter, `$Moves`, which counts one up for every move taken by the piece and checking if this is larger than one.

Requirements 7-16: Are all accomplished.

Requirement 17: The requirement states: "The created board games must be playable in a graphical simulator". We have provided an API that our simulator builds on top of, proving that this is possible. If one wished to create a graphical interface for playing board games in Android, they would simply need to manipulate the object returned by our API, just like our simulator does.

Requirement 18: The requirement and its sub points are not accomplished. Requirement 18a and 18 state that "It must be possible to save the move history" and "It must be possible to start a game from a saved move history". In the right hand side of our simulator, the game info widget is located, displaying the current moves taken other information. So the requirements could be accomplished indirectly by saving this history in a screen shot for instance, restart the game and then perform all the moves that one wants to keep again. This is a clumsy and very manual solution. A better solution would be to save the game in its current state as a file for every

action taken. Nothing hinders this solution from being accomplished, the solution has been down-prioritised. Requirement 18c that stated: "It must be possible to play over a network" is a "could-have" feature, which has also been down-prioritized.

Requirement 19: The requirement states that "The programmer must be able to implement board games with relatively few lines of code". It is difficult to make a quality comparison with other programming languages, since two of the board games are invented by group members and therefore haven't been programmed in other languages. Noughts and Crosses and Connect Four are however well-known board games and these were created in around 20 - 25 lines of code, which we consider as relatively few lines of code.

Requirements 20-24: Are all accomplished.

Requirements 25-27: These requirements can not be evaluated as they are merely a listing of project limitations.

Requirement 28: The requirement states that "The programming language must make it easy and quick for programmers to develop board games". In section 6.1.2, we describe how a game of Connect Four was created in approximately 5 minutes. The speed of which a game can be created in a programming language depends on a number of parameters. First of all, the programmer's familiarity with the programming language and the programmer's experience with a programming mindset. Also, the programming language's support for relevant functionality and its overall writability is a parameter. The fact that we were able to create a fully functional Connect Four game in 5 minutes can be considered as quick and supports that it must be easy to develop board games in JUNTA.

Requirement 29: The requirement is accomplished. Since all of the modules (the scanner, parser, scope checker, interpreter, etc.) of JUNTA are compiled to Java byte code, and hence every system supporting the Java virtual machine is able to run JUNTA games. The simulator on the other hand is implemented using a Java library, that only works on personal computers due to the graphics library used (openGL).

Chapter 7

Conclusion

The goal of the project was to define, design, and implement a programming language. We have accomplished this goal with the creation of JUNTA – a purely functional object-oriented programming language in which it is possible to program board games. In the process, we have defined, designed, and implemented scanner, parser, scope checker, and an interpreter. Furthermore, we have implemented a simulator, which through a game abstraction layer communicates with the interpreter, and in which it is possible to visualise and play JUNTA games in.

In the analysis chapter (chapter 2), we have accounted for different techniques for constructing compilers and interpreters, and further analysed the phases they consist of. In the design (chapter 4) and the implementation (chapter 5) chapters, we have documented formally and informally how we have designed, defined, and implemented JUNTA.

In chapter 3, we formulated a list of requirements which JUNTA and the final solution should be able to fulfil. In section 6.3, we evaluated the requirements and the vast majority of them have been fulfilled. Unfortunately, it was not possible to implement Chess, due to the special rules of the game. The "En Passant" and "Castling" moves turned out to be more difficult to implement than we had expected. We still believe that JUNTA provides the features necessary to implement the game.

Several well-known board games have already been programmed in JUNTA e.g. Noughts and Crosses and Connect Four. The games were created in 19 lines of code and 22 lines of code, respectively (including cosmetic line breaks). We think this clearly shows that it is possible to program board games in JUNTA using only few lines of code.

7.1 Discussion

In this section we will discuss what we could have done differently throughout the project and also discuss future expansion possibilities for JUNTA.

7.1.1 Alternative integer and decimal data structures

Our only primitive data type to represent numerals with, `Integer`, corresponds to Java's primitive data type, going by the same name. We do not however have an alternative for representing decimal numbers or for representing arbitrary-precision integers, if for instance a number exceeds what's possible to represent with a 32-bit signed two's complement integer (Java's specification). It is imaginable that a programmer of a board game would want to use decimals or very big numbers, and therefore implementing this in a later version of JUNTA could be an idea. A possible solution could be a sort of implementation like **BigInteger** and **BigDecimal** data types of Java, maybe even using these as underlying data structures. The **BigInteger** data type provides analogues to all of Java's primitive integer operators, exactly the same

as the primitive data type **Integer**[15], but at the same time it can represent arbitrary-precision integers. That is integers of any size limited only by the memory of the computer. The same holds for **BigDecimal**s.

Decimals do not exist in JUNTA, due to the fact that from all of the board games we've analysed and discussed, not one uses floating point numbers to represent any aspect of the game. We have also chosen not to use either of these, because we do not see a specific need to use such massive numbers. And given that our JUNTA implementation is already slow, using these arbitrary-length data strutures will only bog it further down. It would be rather simple if we ever wished to implement floating point numerals, we would need to update the **isDigit** method in the Scanner to continue picking up numbers when seeing a period. The parser would then need to create a new type of node for it, feeding it to the newly updated **Interpreter** that builds a new type when seeing a floating point node (granted, the operations and types we wish to support floating points on would need to be updated too).

7.1.2 Random values

In JUNTA, the type of games that can be implemented are limited by the fact that randomness is not supported. A game like minesweeper cannot be implemented with randomly generated levels. The board game monopoly would need randomness to implement a good die. A random card cannot be drawn, and a deck of card cannot be shuffled. In our game environment, described in section 4.6.2, many board game related types are described, such as constants and functions. Here it would be natural to also include types like a die type, a card type, and most importantly, a type that can generate random numbers. If the random type would generate random numbers based on a seed, some kind of system time constant should also be available in JUNTA. Another approach would be to automatically seed the random type based on a system time constant every time a game is loaded.

Randomness is not implemented in JUNTA because it is not a functional language trait, since the function wouldn't always return the same value. Hence an implementation with seeding described above could be a possible solution.

7.1.3 Stronger action system

The actions provide everything needed for working with pieces, however when new artefacts are added, actions which effect those are needed. An action to take a card from a deck, an action to play a card, etc. Some artefacts could be generalized: a card could be a `Piece` and a deck could be a `Square`, where the order of multiple pieces is given significance. Generalizing should be done with care though, as over-generalizing will be counter productive with the idea of using a domain specific language. However if cards were pieces, new action types would not need to be created and learned by the user.

Instead of generalizing the artefacts, we could generalize actions. For example, instead of `AddAction` adding a `Piece`, `AddAction` could add any type and `MoveAction` could move both pieces and cards. However now `MoveAction` can depend on an `AddAction` having created an instance of a type, which is not expressed well in `ActionSequence`.

7.1.4 Board types

The game environment of JUNTA makes it easy for programmers to create grid-shaped boards, which is a very basic kind of board defined by a width and a height filled in with squares. But what if the programmer wanted to create a circular kind of square or a hexagonal square? This is currently not possible, but it would be practical in a future version of JUNTA. A possible solution would be a graph representation of the board, where the edges represent squares and transitions represent the border between squares. This would allow greater expressibility and technically allowing every kind of imaginable board, even those expanding over time, such as Carcassone.

Patterns, as they are implemented now, only work on grid-shaped boards because programmers can't define custom `Direction` types. To support all kinds of boards, patterns would have to be able to support these custom `Direction` types. `Direction`s would have to be rewritten so that it'd be possible to define how a certain square is relative to another (i.e. the `Direction` would be the label on an edge going from one square to another).

7.1.5 Efficiency of pattern matching

The design of pattern matching is undoubtedly a strong mechanism for describing moves, win condition, or any other check that depends on a particular board layout. Unfortunately, our implementation of the pattern matching is highly

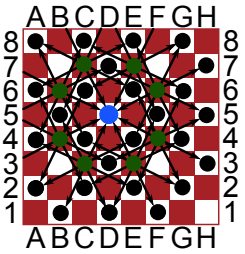


Figure 7.1: *An inefficient implementation.*

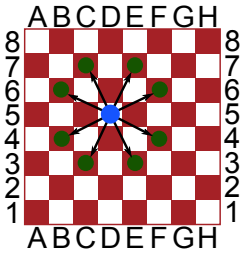


Figure 7.2: *An efficient and intuitive implementation.*

inefficient. Consider the piece placed at the square D5 in figure 7.1. The blue piece can make the moves of a knight from a chess game. This can be specified in JUNTA with the pattern `/(n n e|w) | (s s e|w) | (w w n|s) | (e e n|s) this/`.

With the current implementation, to find the moves of the piece on D5, the pattern must be matched on all 64 squares, given D5 as input. Those squares for which the pattern matching returns true, are those squares the piece can move to. In figure 7.1, only 8 of those 64 checks performed are depicted. For a game of chess starting with 32 pieces, the pattern matching is actually done on all 64 squares (for all 32 pieces). If we for simplicity assumed that each piece had 8 possible moves, the amount of work related to pattern checks for the first move in chess can be calculated to be $32 * 64 * 8 = 16{,}384$. Or more generally, $\mathcal{O}(p * n * m * c)$, where $p =$ the number of pieces, $(n, m) =$ the size of the gridboard, and $c =$ the complexity of each move. It is easy to see how inefficient this approach is, so lets now consider a better and more intuitive approach.

Consider the green piece placed at the square D5 in figure 7.2. The 8 arrows show the moves a knight can make. An easy way to find these squares is simply to start at the knight's square (D5), move two squares in on of the following directions: north, south, east, or west, and then one square in an orthogonal direction. This also seems like a quite efficient approach. This can implemented by modifying the pattern matching to take a square as input and return a list of squares that satisfy the given pattern. A pattern for the knight's move could look like `%this (n n e|w) | (s s e|w) | (w w n|s) | (e e n|s)%`. Notice that the % . . . %-encapsulation is used to distinguish between this modified pattern matching mechanism from the actual pattern mechanism used in JUNTA, which encapsulates a pattern between the dots. This modified pattern matching done on the square D5 would return a list containing the green squares in figure 7.2, namely the legal moves of a knight on D3.

Compared to the current pattern matching in JUNTA, this approach will have the complexity $\mathcal{O}(p * c)$. The amount of work related to the first move of a game of chess would be $32 * 8 = 256$, if we again suppose all pieces are knights. 256 is much better than 163,840 from the previous example. The complexity here seems to not depend on the actual size of the gridboard, but this is only true in some cases, e.g. when considering the moves of a knight. If the the moves of a rook are considered, its constant c will depend on the size of the board for both pattern matching techniques. This is because a larger board will increase the amount of squares the rook can slide to. Generally, patterns containing the pattern-value `*` or `+` can have a complexity that depends on the size of the board.

7.1.6 ADDITIONAL FUNCTIONALITY WITH PATTERN MATCHING

Many different functionalities could be included in the pattern matching. You may have noticed the row of black squares in the bottom of the Connect Four game in figure 6.2. These black squares are put there so the pattern check can allow a piece to be dropped one square north of these squares. If a pattern keyword `outofboard` existed that matched a square outside the board, these squares would not be necessary. A shortcut could also be considered for specifying patterns of the form `/(e3) | (w3) | (s3) | (n3)/`. The `3` is common, but cannot be used outside parentheses like `/(e|w|s|n)3/`, as this means `/(e|w|s|n) (e|w|s|n) (e|w|s|n)/`.

Bibliography

[1] Stig Andersen. Den gode kravspecifikation. *None*, page 3, May 2006. http://www.infoark.dk/article.php?alias=kravspec.

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

[3] Charles N. Fischer. Introduction to Programming Languages and Compilers. ..., Spring 2006.

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

[5] docs.oracle.com. Throwable. http://docs.oracle.com/javase/6/docs/api/java/lang/Throwable.html. (2013-05-24).

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

[7] 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).

[8] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Incorporated, August 2006.

[9] Hans Hüttel. *Transitions and Trees*. Cambridge University Press, 2010.

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

[11] Kurt Nørmark. Overview of the four main programming paradigms. http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html, July 2010. (23-04-2013).

[12] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer, 2010.

[13] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. http://research.microsoft.com/en-us/um/people/emeijer/papers/rdl04meijer.pdf. (07-05-13).

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

[15] Oracle. Class BigInteger. http://docs.oracle.com/javase/6/docs/api/java/math/BigInteger.html, May 2013. (27-05-2013).

[16] Perl. Regular expressions. http://perldoc.perl.org/perlre.html#Regular-Expressions, may 2013. Seen 26-05-13.

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

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

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

[20] Slick Developers. Slick2d. http://www.slick2d.org, April 2013. (20-04-2013).

[21] Sun Microsystems. JavaCC. https://javacc.java.net/, May 2013.

[22] Wouter Swierstra. What are attribute grammars? http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue4/Why_Attribute_Grammars_Matter#What_are_attribute_grammars.3F, July 2005. (30-04-2013).

[23] Parr Terence. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2013. ISBN: 978-1-93435-699-9.

[24] Ronald L. Rivest Thomas H. Cormen, Charles E. Leierson and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2rd edition edition, 2009.

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

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

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

[28] Wikipedia. Parse tree. http://en.wikipedia.org/wiki/Parse_tree, april 2013. (14-04-2013).

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

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

Appendix

APPENDIX A

Abstract Node Types

In section 5.3 we presented a few examples of how we have implemented the grammar described throughout chapter 4. Here we present the rest of the implementations of these abstract node types (ANTs).

A.1 Program structure

The following sections present the structure of programs. The sections are structured as follows:

- Program
- Definition
- Constant definition
- Type definition
- Type body and member definition
- Abstract definition
- Variable list

The ANT for program was presented in section 5.3.

A.2 Definition

Programmers have the opportunity of defining constants and types in their programs. The parser knows that the following piece of code is a constant when it meets the keyword `"define"`, and that the following piece of code is a type when it meets the keyword `"type"`. These two types of definitions will be illustrated with their productions in the following two sections. The production for a defintion is as follows:

$$\begin{array}{lcl} definition & \rightarrow & constant\_def \\ & | & type\_def \end{array}$$

We do not present a figure to illustrate the AST for this production. The production says that a defintion can either be a defintion of a constant or of a type. But the program node says that it is possible to have multiple definitions, so in fact the definitions of constants and types can be interleaved.

A.3 Constant definition

The first type of definition is that of a constant. The production for a this definition is as follows:

$$constant\_def \quad \rightarrow \quad \texttt{"define"}\ constant\ [\ varlist\ ]\ \texttt{"="}\ expression$$

Every definition of this type begins with the `"define"` keyword. The next token must be a constant and this constant can be followed by a list of variables, which is optional. Then an expression is assigned to the constant.

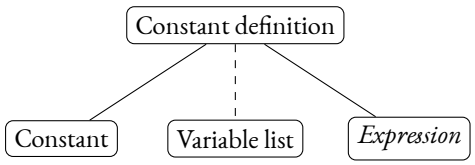


Figure A.1: *The abstract syntax tree for the constant definition node.*

Figure A.1 illustrates the ANT for the constant definition. In the grammar presented above the production also consists of a keyword and an assignment symbol. These are not shown in the ANT because they are not part of an AST (but they are part of a parse tree). The optional node is illustrated with a dashed edge from the parent.

A.4 Type definition

The second type of definition is that of a type. The following production specifies this:

$$type\_def \quad \rightarrow \quad \texttt{"type"}\ type\ varlist\ [\ \texttt{"extends"}\ type\ list\ ]\ [\ type\_body\ ]$$

The parser knows that the following is a type definition by reading the `"type"` keyword. The production says that a type definition can inheret from another type by the use of the `"extends"` keyword. Furthermore, it is possible to have a type body which is explained further in the following section.

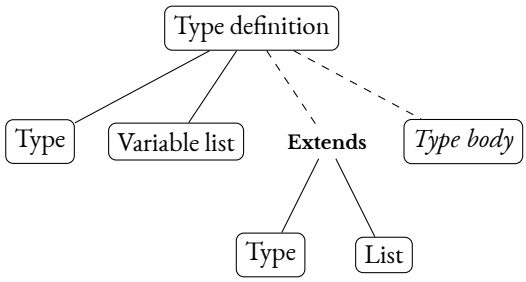


Figure A.2: *The abstract syntax tree for the type definition node.*

Figure A.2 illustrates the ANT of the type definition. This ANT consists of a type followed by a variable list. After the variable list there are two optional choices. The first choice is to inheret from a super type, and this is done by declaring the type of the super type followed by a declaration of a list. The list following the super type is not optional. We have illustrated this by adding a "**Extends**" node as optional folllowed by the two child nodes which are not optional any more if the extends keyword is present. Our implementation of these child nodes is not implemented with an extra node as we illustrate in figure A.2, but it is simply defined in the code that if the keyword is present then the next input must be a type followed by a list. The second choice is to include a type body. These two choices are independent of each other.

A.5 Type body and member definition

In this section we have coupled two productions because they are intertwined and we have chosen not to visualise the ANT of the member definition because it is simply a root with one child. The production of a type definition and the definition of a member is as follows:

$$\begin{aligned} \textit{type\_body} &\rightarrow \texttt{"\{"}\ \{\textit{member\_def}\}\ \texttt{"\}"} \\ \textit{member\_def} &\rightarrow \textit{abstract\_def} \\ &\mid \textit{constant\_def} \end{aligned}$$

The type body begins with a left brace and ends with a right brace which encapsulates the entire code of the type body. Within these braces it is possible to have zero, one, or more member definitions. The code which consists of a member definition can be either an abstract definition or a constant definition. As it was in the program production, it is also possible to interleave the two definitions in this production.

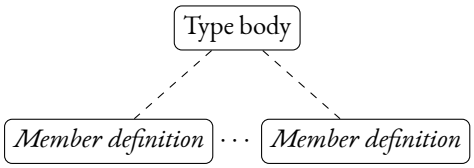


Figure A.3: *The abstract syntax tree for the type body node.*

Figure A.3 illustrates the ANT of the type body. It consists of a root and zero, one, or more member definitions.

A description of the constant definition was presented earlier in this section and the description of the abstract definition will follow this section.

A.6 Abstract definition

The definition of an abstract constant is similar to the definition of a "regular" constant. There are two differences. The first difference is that following the `"define"` keyword another keyword must appear, namely the `"abstract"` keyword. The second difference is that there is no expression assigned at the end of the production, thus being abstract. The following production presents the production from the grammar:

$$\textit{abstract\_def} \rightarrow \texttt{"define"}\ \texttt{"abstract"}\ \textit{constant}\ [\,\textit{varlist}\,]$$

Figure A.4 illustrates the ANT of this production.

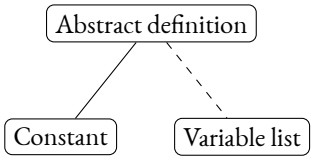


Figure A.4: *The abstract syntax tree for the abstract definition node.*

In figure A.4 the only noticeable difference between the constant and a definition is that there is no expression as the last child. This is due to that the ANTs do not have keywords and such as child nodes.

A.7 Variable list and variable arguments

Within this section we have couple two productions because they are intertwined. The productions are presented in the following grammar:

$$
\begin{aligned}
varlist &\rightarrow \texttt{"["}\ [\, variable\ \{\texttt{","}\ variable\}\ [\, \texttt{","}\ vars\,]\ |\, vars\,]\ \texttt{"]"} \\
vars &\rightarrow \texttt{"..."}\ variable
\end{aligned}
$$

The production specifies that a variable list begins with an occurence of a left bracket and ends with an occurence of a right bracket. Everything between these two brackets are optional, which means that a variable list can be empty.

The last part of the production is a bit hard to read, but it says that if the list begins with one or more variables separated by commas then these variables can be followed by a final comma and the variable argument (vars) which is a list of parameters. If there are no variables then the last variable argument can be included. So, the production consists of two parts - one part containing zero, one, or more variables possibly followed by an occurence of variable argument or just a single variable argument.

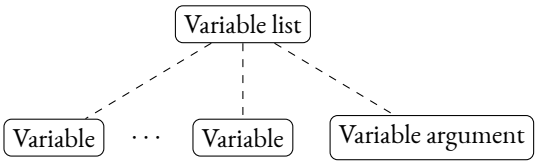


Figure A.5: *The abstract syntax tree for the variable list node.*

Figure A.5 illustrates this by having a root and three children with dashed lines from the root. But this is actually a bit misleading because of the optional parameters in the specification of the production. The production specifies that if the programmer does not want the list to be empty then at least one occurence of a variable must be present or a single variable argument. After the first variable it is possible to have zero, one, or more variables following the first one and another optional variable argument. The easiest way to illustrate this is as it has been done in figure A.5.

This concludes the sections presenting the structure of programs.

A.8 Expressions

The following seven sections present the different expressions in Junta. The sections are structured as follows:

- Operations with precedence and negation
- Element
- Member access
- Call sequence
- Assignment
- If expression
- Lambda expression
- List
- No-operator

The ANT for operations and assignments were presented in section 5.3.

A.9 Element

The following grammar presents the production for an element:

$$
element \rightarrow call\_sequence\ \{member\_access\}
$$

An element begins with a call sequence followed by zero, one, or more member access. This is illustrated differently in the ANT in figure A.6.

Figure A.6 illustrates that the ANT for an element expression will only be constructed if there is at least one occurence of member access, otherwise it will fall through to call sequence and the element node will not be

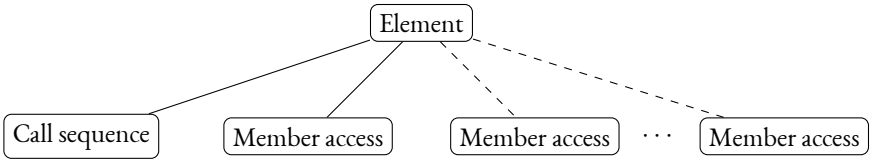


Figure A.6: *The abstract syntax tree for the element node.*

construted. As mentioned earlier in the section explaining operations in JUNTA, this is done to avoid long paths in the AST with single child nodes.

A.10 MEMBER ACCESS

The member access begins with the `"."` symbol followed by a constant and zero, one, or more lists. This is presented in the following production:

$$member\_access \quad \rightarrow \quad \texttt{"."}\ constant\ \{list\}$$

Figure A.7 illustrates the abstract node type for this production. It corresponds exactly to the production.

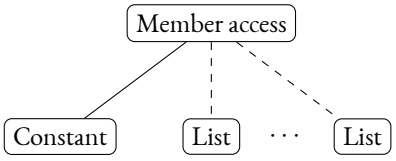


Figure A.7: *The abstract syntax tree for the member access node.*

A.11 CALL SEQUENCE

A call sequence consists of an atomic value, which can be many expressions, followed by zero, one, or more lists. The following production presents this:

$$call\_sequence \quad \rightarrow \quad atomic\ \{list\}$$

We have illustrated the ANT in figure A.8. It does not correspond exactly to the production specified above.

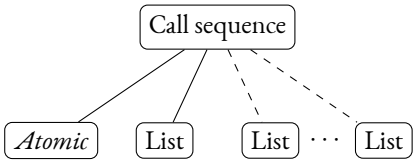


Figure A.8: *The abstract syntax tree for the call sequence node.*

Once again, figure A.8 illustrates that this ANT must consist of at least one list following the atomic value if it is to be constructed. Otherwise, it will fall through and only construct an atomic node. This is due to the same reason as earlier, to avoid single-child nodes.

A.12 If expression

The if expression is specified in the following production:

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

An if expression begins with the keyword `"if"` and ends with the keyword `"else"` followed by an expression. In between the beginning and the end the keyword, `"then"` appears as a must in the if expression. It is for instance not possible to omit the `"else"` statement. The ANT for this production is illustrated in figure A.9 and consists of a root with three children which are expressions. The children are connected to the root with solid lines which means they cannot be omitted.

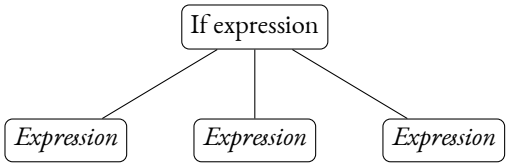


Figure A.9: *The abstract syntax tree for the if expression node.*

A.13 Lambda expression

The lambda expression is specified in the following production:

$$lambda\_expr \quad \rightarrow \quad \texttt{"\#"}\ varlist\ \texttt{"=>"}\ expression$$

Any lambda expression begins with the `"#"` symbol which makes it clear for the parser that this is a lambda expression. A lambda expression consists of a variable list and an expression seperated by the `"=>"` symbol.

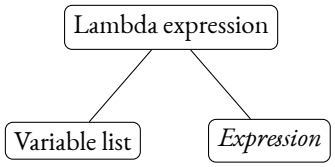


Figure A.10: *The abstract syntax tree for the lambda expression node.*

Figure A.10 illustrates the ANT for this expression. Because a lambda expression is an expression it is possible to have lambda expressions inside other lambda expressions, so it is possible to nest these expressions.

A.14 List

We also have an ANT for lists. Any list can be empty just like a variable list but instead of containing variables a list contains expressions. It can consist of either zero, one, or more expressions. The specification for the abstract node type is specified in the following production:

$$list \quad \rightarrow \quad \texttt{"["}\ [\ expression\ \{\texttt{","}\ expression\}\ ]\ \texttt{"]"}$$

A list begins and ends like a variable list with brackets with optional expressions as the elements of the list. The ANT for this production is illustrated in figure A.11.

The ANT in figure A.11 illustrates that a list can contain zero, one, or more expressions.

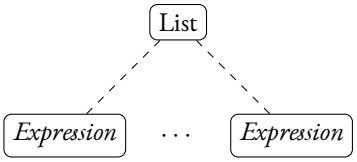


Figure A.11: *The abstract syntax tree for the list node.*

A.15 Not-operator

We have implemented a not-operator that has to do with expressions. The programming language also conists of a not-operator which has to do with patterns. These opreators are not the same. The operator mentioned here cannot be used with patterns.

This operator is not illustrated because it just derives an expression. So, it has one root and one child.

A.16 Patterns

The following sections are concerned with patterns and operators. A pattern consists of a single pattern expression followed by zero, one, or more pattern expressions. This is specified in the following production:

$$pattern \quad \rightarrow \quad pattern\_expr\ \{pattern\_expr\}$$

This production is very straightforward, a pattern must contain of at least one pattern expression. The ANT for this production is illustrated in figure A.12. This ANT has one solid line to the first pattern expression followed by two dashed lines to two other pattern expressions illustrating that these can occur zero, one, or more times.

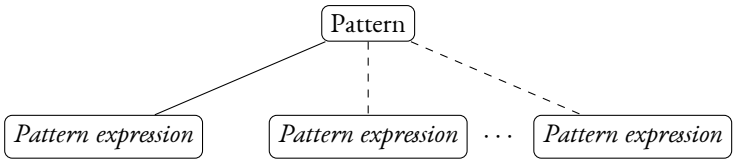


Figure A.12: *The abstract syntax tree for the pattern node.*

A.17 Pattern operators

Junta consists of three different pattern operators. We have an or-operator, a multiplier-operator, and a not-operator.

The or-operator derives a pattern value followed by a pattern expression that both can be many different things. This is why they are written in an italic font. This is illustrated in figure A.13.

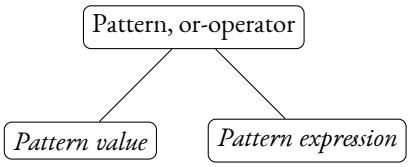


Figure A.13: *The abstract syntax tree for the pattern or-operator node.*

The multiplier-operator derives a single child namely a pattern value. This is not illustrated because it is just a root with one child.

The not-operator also derives a single child namely a pattern check. This is not illustrated because it is just a root with a single child.

A “pattern value” and a “pattern check” can derive different things and would be illustrated with italic font.

This concludes the description of our implementation of the different ANTs which are reflected in the productions of the grammar presented throughout chapter 4.

APPENDIX B

THE CONTEXT-FREE GRAMMAR

This appendix presents the full context-free grammar of JUNTA

Program structure:

$$
\begin{array}{rcl}
program & \rightarrow & \{definition\} \\
definition & \rightarrow & constant\_def \\
 & | & type\_def \\
constant\_def & \rightarrow & \texttt{"define"}\ constant\ [\ varlist\ ]\ \texttt{"="}\ expression \\
type\_def & \rightarrow & \texttt{"type"}\ type\ varlist\ [\ \texttt{"extends"}\ type\ list\ ]\ [\ type\_body\ ] \\
type\_body & \rightarrow & \texttt{"\{"}\ \{member\_def\}\ \texttt{"\}"} \\
member\_def & \rightarrow & abstract\_def \\
 & | & constant\_def \\
 & | & data\_def \\
abstract\_def & \rightarrow & \texttt{"define"}\ \texttt{"abstract"}\ constant\ [\ varlist\ ] \\
data\_def & \rightarrow & \texttt{"data"}\ variable\ \texttt{"="}\ expression \\
varlist & \rightarrow & \texttt{"["}\ [\ variable\ \{\texttt{","}\ variable\}\ ]\ [\ \texttt{","}\ vars\ ]\ |\ vars\ ]\ \texttt{"]"} \\
vars & \rightarrow & \texttt{"..."}\ variable
\end{array}
$$

Identifiers:

$$
\begin{array}{rcl}
constant & \rightarrow & (\ lowercase\ \{alphanum\}\ )\ -\ (\ reserved\ |\ direction\ ) \\
type & \rightarrow & (\ uppercase\ \{alphanum\}\ )\ -\ coordinate \\
variable & \rightarrow & \texttt{"\$"}\ alphanum\ \{alphanum\}
\end{array}
$$

Expressions:

expression → *let_expr*
| *if_expr*
| *set_expr*
| *lambda_expr*
| `"not"` *expression*
| *lo_sequence*

lo_sequence → *eq_sequence* {(`"and"` | `"or"`) *eq_sequence*}

eq_sequence → *cm_sequence* {(`"=="` | `"!="` | `"is"`) *cm_sequence*}

cm_sequence → *as_sequence* {(`"<"` | `">"` | `"<="` | `">="`) *as_sequence*}

as_sequence → *md_sequence* {(`"+"` | `"-"`) *md_sequence*}

md_sequence → *negation* {(`"*"` | `"/"` | `"%"`) *negation*}

negation → *element*
| `"-"` *negation*

element → *call_sequence* {*member_access*}

member_access → `"."` *constant* {*list*}

call_sequence → *atomic* {*list*}

atomic → `"("` *expression* `")"`
| *constant*
| *type*
| *variable*
| `"this"`
| `"super"`
| *integer*
| *string*
| *direction*
| *coordinate*
| `"/"` *pattern* `"/"`
| *list*

let_expr → `"let"` *variable* `"="` *expression* {`","` *variable* `"="` *expression*}
`"in"` *expression*

set_expr → `"set"` *variable* `"="` *expression* {`","` *variable* `"="` *expression*}

if_expr → `"if"` *expression* `"then"` *expression* `"else"` *expression*

lambda_expr → `"#"` *varlist* `"=>"` *expression*

list → `"["` [*expression* {`","` *expression*}] `"]"`

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 → `"friend"`
| `"foe"`
| `"empty"`
| `"this"`
| *type*