FELADATKIÍRÁS

Az elektronikusan beadott változatban ez az oldal törlendő. A nyomtatott változatban ennek az oldalnak a helyére a diplomaterv portálról letöltött, jóváhagyott feladatkiírást kell befűzni.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Kurdi Barnabás

Creating a textual language for building deck-building games

Somogyi Ferenc Attila

BUDAPEST, 2022

Tartalomjegyzék

[Abstract 5](#_Toc116292049)

[1 Introduction 6](#_Toc116292050)

[1.1 Deck-building board games 6](#_Toc116292051)

[1.2 Creating a text-based programming language 7](#_Toc116292052)

[1.3 Motivation 7](#_Toc116292053)

[1.4 Overview of rest of the paper 8](#_Toc116292054)

[2 Creating Tabl 9](#_Toc116292055)

[3 An interactive experience 10](#_Toc116292056)

[4 Case Studies 11](#_Toc116292057)

[5 Closing thoughts 12](#_Toc116292058)

Hallgatói nyilatkozat

Alulírott **Kurdi Barnabás**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 10. 10

...…………………………………………….

Kurdi Barnabás

Abstract

Deck-building games have been a staple board game genre ever since the release of *Dominion* in October 2008. In these games players start with a weak set of cards as a personal deck and seek to strengthen their deck with cards from a predetermined array of options, competing against others in the process for victory. No option to date exists for the easy construction and testing of these types of games from scratch in a text-based format.

*Tabl* is a text-based programming language designed to alleviate the aforementioned problem for people who are adept at making deck-building games and curious individuals who want to try their hand at it alike, all while requiring little to no previous programming knowledge. The language comes with its own interpreter written in Python and allows the users to try the card game they have made in a web-based graphical environment developed in Vue.js 3. A file written in *Tabl* is meant to resemble the rulebook of the game being played.

The language is made with further expansion in mind, leaving the possibility of additional development to broaden the scope of it to encapsulate other genres of games.

# Introduction

## Deck-building board games

Board games come in many different shapes and forms: some people enjoy the thrill of rolling the dice for a chance to win big in *Monopoly*, while others prefer crushing their opponents mentally in purely skill-based, deterministic games like chess. These two styles provide a basis for a possible categorization of modern board games. Board-game fans call games that rely more on the luck of the draw and the theatrics of play American-style (or *Ameritrash*), while they dub games that are more abstract and require careful planning and thought throughout *Eurogames*. (examples?)

These two types have very different skill-testing elements. Games that fall into the *Ameritrash* category generally require players to be quick on their feet and fast to adapt to a wide array of situations. *Eurogames* however reward long-term planning and a distinct type of abstract or mathematical thinking. Naturally, most board games borrow different elements of the two categories and mix them together to provide the best play experience possible. One excellent example of this is the genre of deck-building board games (deck-builders).

The main idea behind deck-builders is the following: what if players were subject to the luck of the draw, but had a chance to tip the scales in their favor by manipulating their own deck during play? The concept of collectible card games (CCGs) is very similar, however the decks used are constructed before play begins, while in deck-builders deck construction is the focus of the main gameplay loop. The first game to fully embrace this idea was *Dominion,* released in 2008.

Most deck-builders follow a very similar play pattern. All of the players start with a pre-constructed personal deck of weak cards that don’t have any substantial effect on their own. Players take turns draw a number of these cards, playing them to gain per-turn resources that they can use to buy cards from a common supply. The played cards and the ones players buy during their turn go into the players’ personal discard pile. Once a player’s personal deck is depleted, they shuffle their discard pile to form a new deck of cards to draw from, with their decks now containing the cards they have previously bought.

These cards can have a wide variety of effects. Some of them provide the player with economical advantages, allowing them to buy more cards during subsequent turns. Others make it possible to draw more cards from the deck than normal, allows players to discard cards in order to draw new ones, or enables them to remove the weaker cards from their deck permanently. Several cards facilitate player interaction: making others discard their own cards or stealing resources from them.

The end goals of these games can differ greatly with some requiring the players to accumulate the highest number of victory points. In others, players battle until one of them loses all of their health points. However, no matter the objective, the necessity of players to improve their decks bit by bit remains a constant.

## Programming languages

The number of programming languages used in the world is ever-increasing. General-purpose programming languages like C, highly specialized ones only used in certain fields like R, educational programming languages that are used to teach programming to children, such as Logo or Scratch and esoteric languages like Shakespeare which are created not for commercial use, but to test the limits of programming languages and to have fun of course.

Programming languages can be graphical, providing a graphical interface for users to interact with the language, making software creation easier to understand and omitting the need for users to learn the sometimes-difficult syntax of programming languages. The other, more common type of programming languages is textual or text-based programming languages, these give users a bigger degree of freedom when creating software, however usually they require more finesse to utilize fully.

In this thesis I will be focusing primarily on the creation of textual programming languages, detailing the basic principles, and taking a look at a case study of a language I have made.

## Motivation

I have always been fascinated and deeply entertained by board games. They provide a safe environment to test one’s mettle against their friends in a competition of wits. After I was introduced to deck-builders I was highly intrigued: they combine luck and strategy in a very distinct way. Deck-builders don’t allow players to calculate their way to a certain victory, requiring critical and strategic thinking in a more meta way. One might not be able to decide what cards they draw in a turn, but they can influence the pool of cards they draw from.

Creating my very own programming language has been a dream of mine ever since I learned of esoteric programming languages like *Shakespeare* or *Whitespace* when I first became interested in coding. The task seemed daunting and a bit too complex for me just to dip my toes into it with no clear goal in mind. Was making a funny or interesting language just for the sake of it worth the time and effort?

I realized that there was a way for me to combine my passion for board games (specifically deck-builders) and my interest in creating my own language. What if there was a way for anyone to create their own deck-building game and test it in a graphical environment online? This is how I came up with the idea of *Tabl.* My goal with Tabl was not only to create a language that could define a deck-builder (it might be possible to do so with the use of a data interchange language like JSON, although more dynamic elements of the game might be harder to define), but to make the experience of creating one easy to understand and to structure the language in a way that helps facilitate the ease of translating it into a playable experience. I also wanted to make the files written in the language to be multi-purpose. Not only would they define the author’s custom deck-builder, but they would also look like a traditional board game rulebook, making it easy for anyone to tell the rules of the created game at first glance.

My aspirations for *Tabl* are two-fold. On one hand I want to make it possible for lovers of the genre with no previous programming experience to try their hand at creating a deck-builder for the first time. On the other, I hope even experienced game designers can find some merit in the concept as well, having a way to effortlessly modify both the minute and bigger, overarching rules of their game and test the changes immediately. This allows for easy evaluation and tweaking of game balance.

## Structure of the thesis

In the next section of the paper, I will highlight and explain in detail the different phases of crating the *Tabl* programming language and the interpreter I have developed to process files written in the language. In the section after, I am going to focus on the technological aspects that make testing the created deck-building game possible, describing the different technologies used in both the back- and front-end of the web application.

The third section deals with two case studies: I have created pre-existing card games *Dominion* and *Hero Realms* in *Tabl* and I will be comparing playing these deck-builders through the web application and in real life, testing the product’s viability.

The last section contains my closing thoughts about the project and discussion about the possibility of the expansion of its scope to other types of board games besides deck-builders, or maybe even to board games in general.

# Background and technologies

## A brief history of programming languages

With the creation of computers arose the need for programming languages. At first, in the 1940’s, programmers had to code using a sequence of 0’s and 1’s, essentially writing machine code. Only low-level operations like data manipulation and simple arithmetic were possible, and writing a program was hard and tedious. The programs written were extremely hard to understand and neigh impossible to modify, being written most of the time on punched cards – once a programmer made a mistake, they most likely had to completely restart.

The first wave of user-friendly languages came with the creation of assembly languages in the early 1950’s. These languages allowed for an easier understanding of machine instructions and later made it possible for programmers to define early variations of functions.

Then, in the late 50’s came a boom of higher-level, specialized programming languages that were created with further ease of use in mind, like *Fortran*, *Cobol*, or *Lisp*. *C*, maybe the most widely known and still one of the most-used programming languages was created in the early 70’s, first being used to create Unix utilities, then slowly gaining a wider appeal, with compilers available for many modern operating systems and architectures.

Today, more than a thousand programming languages that are in use exist. Modern languages differ greatly from early ones, emphasizing problem-solving in particular areas like web development, data analysis or data management and providing users with an ever-increasing array of robust tools and high-level built-in functions that make writing and understanding languages easier. Despite these differences, one aspect of programming has not changed ever since the beginning: programs written in any language need to be translated to machine code to be executed. The software that achieves this translation is called a compiler.

## Compilers and interpreters

As stated previously, compilers are an integral part of programming languages, since they are needed for the code to be translated to machine code, understandable to a computer. Compilers take a source program (usually written in a higher-level language) and produce a target program written in machine code. This target program then can be ran by the end-users.

Interpreters are very similar to compilers, however instead of translating the source program into a target program, they execute the operations written line-by-line, producing an output without the need of a separate target program to be run. Interpreted programs allow for easier debugging due to the line-by-line execution, however started out being slower than compiled programs.

JIT (Just-in-time) or run-time compilation combines the advantages of compilers and interpreters, translating code first into an intermediate bytecode, then translating bytecode into machine language. This allows for the faster run-times of compilation. while ensuring the ease of debugging and code analysis of interpreted languages.

Although these approaches differ in translating a source program written in a high-level language to target machine code, their structure remains largely the same.

### Analysis

The first step of compilation is called lexical analysis or scanning. During this phase, the lexical analyzer (lexer) breaks up the character stream (the program written in a language) into meaningful character sequences called lexemes. For each lexeme, the analyzer produces a token. These tokens are usually made up of two parts: token-names and attribute-values.

Token-names serve to denote what part of a language a lexeme is meant to represent. This could be anything from an identifier to an operation type like addition or assignment. Attribute-names are pointers that point to entries in the symbol table of a compiler. Any lexeme in a program that is created by the user, like the name of a variable is stored in the symbol table.

After lexical analysis comes syntax analysis or parsing. The syntax analyzer puts the tokens created by the lexer into a tree-like structure. This syntax tree carries in it additional information compared to only the tokens: it denotes how the tokens relate to each other grammatically, showing the order of operations and which operands a certain operation is meant to be performed on.

Semantic analysis is the last part of the analysis process. The semantic analyzer is responsible for checking whether the code adheres to the ground rules laid down by a language, pertaining to control structures and data types. Semantic analysis includes (but is not limited to) type-checking and conversion.

*ANTLR (Another tool for Language Recognition) v4* is a parser generator that can be used to create a lexer and parser for any user-defined language. These language definitions are called grammars, containing lexer and parser rules. Parser rules are higher-level representations of concepts found in a grammar, defining what kind of data is to be extracted from a program. Lexer rules are lower level rules that define what kinds of character sequences are acceptable in certain parts of a parser rule. Parser rules can contain other parser rules, however the base building blocks any grammar are lexer rules.

*ANTLR 4* supports grammars written in a wide variety of languages (e.g. Java, Python and C#) and can also be used to generate parse-tree walkers in these languages. These walkers can be used to process the programs written in the created language in the language of the grammar.

# The *Tabl* programming language

# Crating a front-end application

# Case Studies

# Conclusion