# Programming Arcade Games using Natural Language

Utilizing inherent language skills as a gentler introduction to Computational Thinking

Bachelor's thesis in Computer Science

Madeleine Lexén, Erik Ljungdahl, Hanna Rydholm,
Henning Sato von Rosen

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Bachelor's thesis 2019

# Programming Arcade Games using Natural Language

Utilizing inherent language skills as a gentler introduction to Computational Thinking

MADELEINE LEXÉN
ERIK LJUNGDAHL
HANNA RYDHOLM
HENNING SATO VON ROSEN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Programming Arcade Games using Natural Language
Utilizing inherent language skills as a gentler introduction to Computational Thinking
MADELEINE LEXÉN, ERIK LJUNGDAHL, HANNA RYDHOLM,
HENNING SATO VON ROSEN

Supervisor: Peter Ljunglöf
Examiner: Wolfgang Ahrendt, Morten Fjeld, Sven Knutsson, Miquel Pericas

Bachelor's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Programming Arcade Games using Natural Language
Utilizing inherent language skills as a gentler introduction to Computational Thinking

MADELEINE LEXÉN, ERIK LJUNGDAHL, HANNA RYDHOLM,
HENNING SATO VON ROSEN

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

## Abstract

Due to the technological advances in society, the need for digital competences is increasing. The purpose of this thesis is to develop a web application that introduces children between ages 10 and 18 to basic programming concepts and train them in Computational Thinking by using their existing knowledge of Natural Languages. By using a restricted part of Natural Language instead of a programming language, we take advantage of the users inherent language skills, with the aim of a gentler introduction to Computational Thinking. Questions that are treated include, among others, what contribution can be made to the education of children in Computational Thinking, how to handle the input from the user, how to transform the input to a playable game and how the system should handle errors in the input. These questions were investigated by the development of an application which consists of a parser, implemented using a third party parsing library (Nearly.js), an Evaluator, which evaluates the result from parsing the input and organises it in a data structure that represents the game. This data structure, the game representation, is then given to a game engine that constructs a playable game according to the users intentions. This game is then displayed in the User Interface, where the user can interact with it. Possible strategies for evaluating our system are A-B testing, survey or user analysis. Furthermore, extensions on the application include more extensive error messages, support for multiple source languages and providing challenges to test the user. We conclude that the result was a modular and extendable application. Moreover, it is our conviction that our application does contribute to learning parts of Computational thinking.

Keywords: Computational Thinking, Natural Language, education

# Sammandrag

De teknologiska framstegen i samhället ökar behovet för digital kompetens. Syftet med arbetet är att utveckla en webbapplikation som introducerar *Computational Thinking* och grundläggande programmeringskoncept för barn i åldrarna 10 till 18 år, genom att uttnyttja deras redan existerande förmågor i naturligt språk. Genom att använda en begränsad del av naturligt språk drar vi nytta av användarens inneboende språkkunskaper, med målsättningen att ge en mjukare introduktion till *Computational Thinking*. Frågor som behandlats inkluderar vilket bidrag som kan göras till att lära barn *Computational Thinking*, hur indatan från användaren ska hanteras, hur denna input ska omvandlas till ett spelbart spel och hur systemet ska hantera fel i inputen, bland andra. Dessa frågor undersöktes genom utvecklandet av en applikation som består av en *Parser*, implementerad med hjälp av ett parsing bibliotek (Nearly.js), en *Evaluator*, som evaluerar resultatet från att läsa inputen och organiserar det i en datastruktur som representerar spelet. Denna datastruktur, spelrepresentationen, ges sedan till en *Game Engine*, som konstruerar ett spelbart spel i enlighet med användarens avsikter. Detta spelet visas sedan för användaren i användargränssnittet, där användaren kan interagera med det. Möjliga strategier för att utvärdera vårt system är A-B testning, undersökning eller användaranalys. Utökningar av applikationen inkluderar mer omfattande felmeddelanden, stöd för mer än ett källspråk och att tillhandahålla utmaningar för att testa användaren. Vi drar slutsatsen att vår applikation är modulär och utbyggbar. Vidare är det vår övertygelse att vår applikation bidrar till att lära ut delar av *Computational Thinking*.

Nyckelord: Computational Thinking, Naturligt språk, utbildning

# Acknowledgements

We would like to thank our supervisor, Peter Ljunglöf. We are humbled and grateful for you patience and support. Not only did you offer gentle guidance into a fascinating subject matter, but you also helped us function together as a group.

Madeleine Lexén
Erik Ljungdahl
Hanna Rydholm
Henning Sato von Rosen

Gothenburg, May 2019

“There need be no real danger of it [writing instruction tables] ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.”

---

*Alan Turing in the report "Proposed Electronic Calculator" (1946), written before the first computer was completed. The expression “writing instruction tables” was used for what we today call “programming”.*

# Contents

# 1
# Introduction

This chapter goes through the background of the project and presents the purpose in the context of the background. Additionally, the methods used and the delimitations of the project are presented

## 1.1 Background

As technology advances and becomes prevalent in society, the need for digital competences and understanding increases. Access to computers and knowledge and how to use them is becoming increasingly more important in today's information society, and is a new source for inequality in society, which is referred to as the *digital divide* [1]. A basic understanding of programming will make it easier for people to operate in our highly technological society.

There are several benefits of knowing how to program and developing what is called Computational Thinking. It evolves analytical thinking, and broadens a person's ability to structure and develop solutions [2]. Teaching students Computational Thinking is a challenge, which schools already have begun to undertake. A 2015 in-depth study on the integration of digital competence and coding in education in European countries found that 16 out of the 21 countries surveyed had already added coding skills in some form to their curriculum [3]. In 2017, the Swedish Ministry of Education Research modified the education curriculum to reflect these changes in society [4]. These changes in the curriculum introduce the need for tools that educators can use to teach coding skills and logical thinking to children.

There is a widespread idea that programming is a skill that one needs to be taught, and something that only a few have a predisposition for, neither of which is correct. At its core, programming is recognising a problem and defining it, and then formulating a solution as a series of instructions. This is something that everyone already knows how to do, using Natural Language. This thesis takes a unconventional approach to teaching programming, by utilising the students inherent knowledge of Natural Language, instead of teaching a specific programming language. By doing this the focus is on the concepts, which helps the student develop their skills in Computational Thinking, rather than a programming language.

Given a system that can interpret a controlled set of Natural Language in a logical and consistent way, it enables the user to quickly understand the environment and develop a more formal and logical way of thinking. This is the thought behind projects like GameChangineer [5], created by professor Michael Hsiao of Virginia Tech. It supports creating a range of simple arcade games using typically between 20 and 30 lines in Natural Language. Our project takes GameChangineer as an inspiration, and will center around the same concepts.

## 1.2 Purpose

The purpose of this thesis is to develop a web application that introduces children between ages 10 and 18 to basic programming concepts and train them in Computational Thinking by using their existing knowledge of Natural Languages.

The intention is to restrict the range of accepted input in Natural Language, i.e. the project will make use of a controlled Natural Language, with clear restrictions on the kind of sentences and expressions the system will understand.

When viewed in a larger context, we strive to increase the digital competence in society with our project, so as to both give the individual more freedom and to lessen the digital divide.

## 1.3 Problem formulation

As mentioned in the previous section, part of the purpose is to develop a web application that can be used as an educational tool when teaching children programming. This application generates playable games given a description in Natural Language. This is a proposal that raises a couple of interesting questions.

Since the purpose is to give an introduction to programming and teach Computational Thinking, a natural question to ask is how to teach this. Furthermore, what skills can reasonably be expected to be learnt in this manner, and how can what the user learnt be evaluated.

Another question is how to handle user input; what kind of input should the system accept, and in what format. This also raises the question how errors in user input should be handled. How lenient should the system be towards mistakes the user might make, and how should these errors be communicated to the user. In addition, how should the system compensate for these errors, if indeed it should compensate for them.

A problem that is especially interesting from a programmers perspective is how to create a game from the input. What data structures should be used to represent the concepts identified in the users input, so that it is possible to generate a game from that representation. This leads to another interesting question, which is how you write code that can represent the users' intentions, before knowing what those intentions are. How should the concepts identified in the user input be represented, in order for them to be usable as a blueprint for the game representation? Granted, there will be some restriction on what will be possible to generate. Still, the system must be able to create game instances that the user then can interact with.

## 1.4 Related work

Since the 1970's, attempts have been made to take advantage of the possibilities of computers to make programming and formal thinking more accessible to people in general, and beginners in particular. Five notable approaches are:

**LOGO**
Developed (1967-) by Seymour Papert et.al. – make a "turtle" move using small composable actions [6].

**SmallTalk**
Developed (1971-) by Alan Kay et.al. – general purpose programming language and environment with messaging and object orientation [7].

**Scratch**
Developed (2002-) by a group at MIT led by Mitchel Resnick. Lets the user create animations and games by drag-and-drop manipulation of blocks of vividly visualised programming language syntax [8].

**GameChangineer**
Created by professor Michael Hsiao of Virginia Tech. A web application that lets the user program simple arcade games using English [5].

**WordsEye**
A web application that lets the uzer use Natural Language to describe a scene which gets generated into a 3D picture [9].

What all these approaches have in common is the presence of an interactive graphical environment that makes it possible for the user to see more or less immediate results from programs or, in some cases, parts of programs. LOGO and SmallTalk also utilise *metaphor*: In LOGO, the user can imagine him- or herself as the character whose moves is being programmed. In SmallTalk, the user sees the program as made up of "small computers" called *objects*, that sends and receives messages and perform actions, not unlike human beings in ordinary life. Scratch on the other hand does not utilise metaphor directly but instead uses visual syntactic elements, not unlike a puzzle game. The bits and pieces fit together if they are compatible, to ease the user into correct grammatical usage of traditional computer programming code. GameChangineer and WordsEye supports Natural Language input as a primary input method but differs from our project in that they allow non-precise interpretations of the user input. WordsEye differs from the other mentioned approaches in that its primary purpose is to let non-specialist users utilise advanced 3D scene generation techniques in a social media context.

## 1.5 Delimitations

The project will not include user testing, instead user stories are employed to evaluate the application. This is due to time limits, as user testing would be useful to measure the outcome of the project.

There is also some restriction on the input language; the application will only be developed with support for English, and it will only be capable of interpreting instructions in English as well.

The application will be a browser-only application, but with desktop browsers in mind, and will thus not be adapted to work on mobile devices.

## 1.6 Overview

In the next chapter we will present theoretical background relevant for understanding the result. This includes a more in depth look at what Computational Thinking is and how it relates to our project Furthermore, we give a walk through of the components used in the project.

The chapter that follows details the methods used when working with the thesis, which consists of both methods of evaluation and for implementation. Then we give a full description of our project, both the thoughts that lie behind and what the outcome of the project was.

In the discussion chapter it is described what impressions and insights the project resulted in. It also discusses what work this could lead to, and how the project could be further developed in the future. The last chapter contains our conclusions based on the result we reached.

# 2
# Theory

This chapter gives a theoretical background to the concepts treated in our project. It then continues to explain the tools and methods used to develop our product, and the theory behind the grammar we created.

## 2.1 Computational Thinking

The idea that computer scientists benefit from learning to think, reason and recognise patterns in a certain way has been around for a long time. In 1960, Alan Peril argued for the usefulness of Computational Thinking as a skill not only useful for computer scientists, but for other areas and problems as well [10]. In 1980, Seymour Papert used the term ‘Computational Thinking’ for the first time in his book *Mindstorm*, where he writes about computers and how thinking and learning may change as they become an everyday tool [6].

The term has been around and discussed within the computer science community ever since. In 2006, however, it was introduced to a larger audience after an article about the subject was released by Wing [11]. Wing suggested that Computational Thinking could be of value to a wider audience than just computer scientists, and should be more widely taught. In a followup article four years later, Wing reference a definition by Wing, Cuny and Snyder where they define Computational Thinking as:

> “Computational Thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent.” [12].

This definition is very broad and open to interpretation, which has lead to problems when implementing the subject in teaching environments [13]. In recent years, the focus has shifted slighly from trying to define the term to defining different components that are necessary for understanding Computational Thinking [14]. In a 2012 paper that focused on bringing Computational Thinking into schools, Barr and Stephons identified what they consider to be the foundation of Computational Thinking: Data collection, data analysis, data representation, problem decomposition, automation, algorithms and procedures, parallelization, and abstraction [15].

Another similar definition was made by the Computer Science Teachers Association (CSTA) and The International Society for Technology. Their definition is as follows [16]:

- Formulating problems in a way that enables us to use a computer and other tools to help solve them.
- Logically organizing and analyzing data
- Representing data through abstractions such as models and simulations
- Automating solutions through algorithmic thinking (a series of ordered steps)
- Identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources
- Generalizing and transferring this problem solving process to a wide variety of problems

Some of these concepts can be hard to teach younger children. Research on how this can be done is still ongoing. Projects like GameChangineer attempt to teach some of these concepts in a simple and fun way.

The studies that exist on teaching Computational Thinking to children are often small in sample size and give varying results. A 2016 study on teaching Computational Thinking to preschoolers by using ScratchJr gave positive results and enhanced the childrens' ability to abstract and compartmentalise [17].

## 2.2 Game Engine

Many of the games developed today make use of a game engine, with the engine being a core part of the game. To give an exact definition of game engines is non-trivial, since there is no formal definition of the concept. However, a game engine encapsulates the general components of a game that can be abstracted and reused. It can be viewed as a set of rules, algorithms and forms, that are applicable to an arbitrary game. Another way to describe it is as a "collection of modules of simulation code that do not directly specify the game's behaviour (game logic) or game's environment (level data)" [18, p.2].

Many games use physics for example, the rules of which are unlikely to change. It is not necessary for every game developer to implement a new physics system, but can instead reuse an already existing one. This saves both time and resources as games are being developed, making the whole process less complicated. Game content, which is specific to a game, is not included in the game engine. This is where graphics and sound resides, things that depend on the specific game being developed [19].

## 2.3 Entity Component System

In game development, entities with several features are generally initiated from a class containing the relevant components of that entity. Creating such a class, that can handle a mix of features, is often solved with Object Oriented Programming (OOP), which is a programming architecture used to make code modular and reusable [20]. In OOP, this is achieved using inheritance. Data and behaviour is encapsulated in classes and objects, which inherit from each other to build representations with the right data and behaviour [21]. This introduces dependencies, which may result in difficulties when creating game objects, since these objects can share features. This can result in an unintuitive inheritance hierarchy, or increase the need for custom handling. As the complexity of the hierarchy increases, the expandability of the code decreases [20].

A very simple example of this can be seen in Figure 1, where a inheritance structure doesn't work for a new class, requiring re-factoring of the inheritance hierarchy. In large code bases, this can be time consuming and confusing. As a solution to this the Entity Component System (ECS) pattern was introduced, which is similar to both the component and the strategy pattern from OOP [22]. The ECS pattern uses composition to provide functionality instead of inheritance. Composition gives objects a 'has-a' relationship, rather than an 'is-a' relationship. Objects are created by adding certain components, making code reusable and objects easily modifiable by removing or adding components [20]. An example of how components are used to dynamically add behaviour to entities can bee seen in Figure 2.

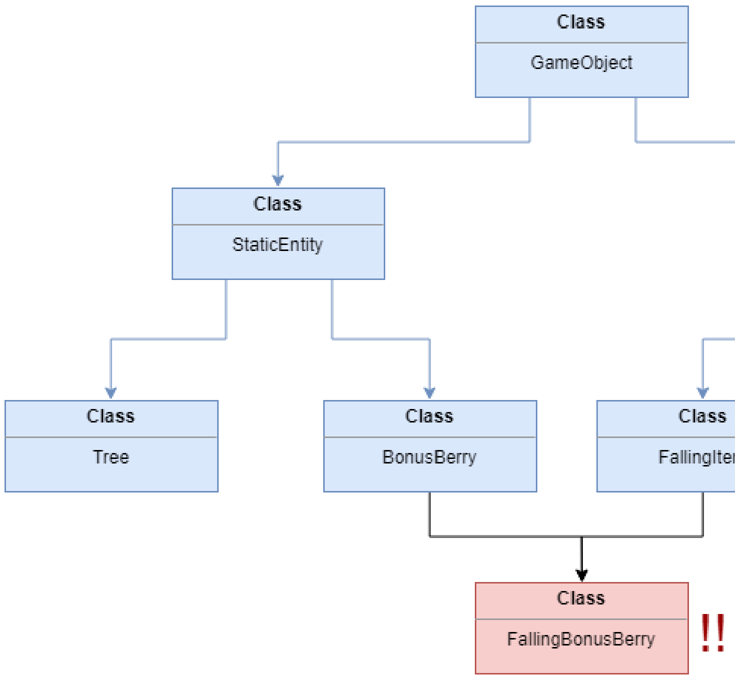


**Figure 1:** Using OOP, the 'FallingBonusBerry' object cannot inherit both the classes 'BonusBerry' and 'Falling', as it would need to inherit from both the "StaticEntity" and "MovingEntity" classes.

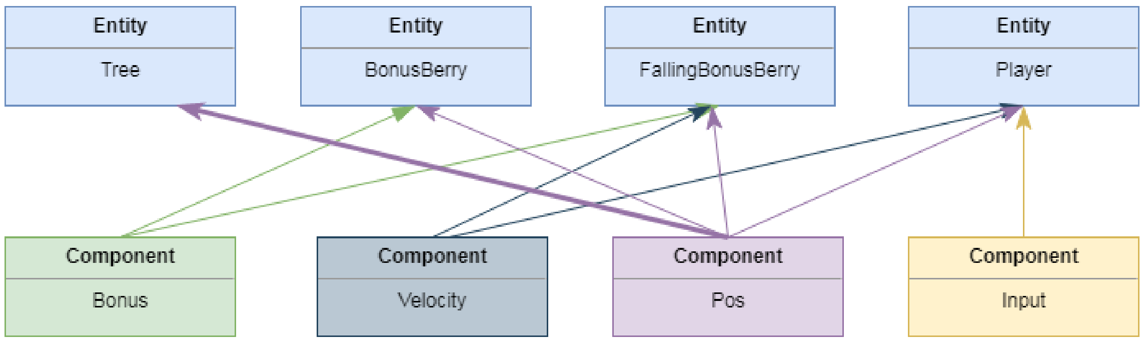


**Figure 2:** In ECS the entity can have several components. In this case the entity 'BonusBerry' is composed of 'Bonus' and 'Pos', while 'FallingBonusBerry' has the components 'Pos', "Bonus" and "velocity".

The ECS pattern is organised in entities, components and systems, where an entity is composed by several components. Entities are objects with unique ids connected to them, which in game development translates to game objects. Different properties of this entity are represented by the components. Lastly, systems are responsible for a part of the simulation, contains the logic for manipulating or utilising the data from the components. Examples of a system is a physics engine or a rendering module. What differentiates this method from traditional OOP is that the ECS pattern uses composition instead of inheritance, as previously mentioned, resulting in highly flexible and extensible frameworks [20][22].

The benefits of ECS is that systems are modular and expandable. Given the isolated nature of systems and components, adding new features becomes easy. This isolation is also the drawbacks of the ECS pattern; cross system communication and shared components are difficult since they go against the decoupled and isolated nature of ESC [20].

## 2.4 Grammar

A context-free grammar (CFG) defines rules for a formal language. A CFG consists of a finite set of symbols called *terminals*; these make up the strings of the language. There are also a finite set of *variables* or *non-terminals*, where each variable represents a language. One of these symbols is called the *start symbol*, and represents the language being defined. Furthermore, the recursive definition of a language is represented by a finite set of *production rules*. Each rule contains a variable, called the head, which is partially defined by the rule. Next, after the head, follows the symbol $\rightarrow$, which in turn is followed by the body of the production; a string of zero or more terminals and variables [23].

Developing a Context Free Grammar that describes a Natural Language is not an easy task. However, when looking at a small, controlled subset of English, it is possible.

# 3
# Methods

This chapter introduces the methods used in this project in order to evaluate the results and to investigate the questions specified in the problem formulation. This was done partly by implementing a Web application, and partly by studying existing research in those areas.

## 3.1 Evaluation

In order to access the success of the project, two methods of evaluation were employed, which are described in more detail below.

### 3.1.1 User stories

User stories were used in order to evaluate the web application, but also as a tool to plan and discuss the project. A user story is a description of functionality that has value to either the systems user or its producer. We have used Mike Cohn's definition which is as follows [24]:

> User stories are composed of three aspects:
>
> - a written description of the story used for planning and as a reminder
> - conversations about the story that server to flesh out the details of the story
> - tests that convey and document details and can be used to determine when a story is complete

### 3.1.2 Test cases

Another evaluation technique used was to define test cases that were run on the application, the result of which were used to evaluate if the application behaves as expected and if the given test case resulted in the expected translation, or the expected game.

The first set of test cases we used were tests developed for our product to support a simple version of the game “Brick Breaker”:

**Brick Breaker**

```
There is a paddle, 54 bricks, and a ball.
The paddle is positioned at the bottom of the screen.
The player controls the paddle.
The paddle can move sideways.
The ball is positioned at the middle of the screen.
The ball starts moving randomly.
The bricks are positioned in a block at the top of the screen.

When the ball hits the paddle, the ball bounces.
When the ball hits a brick, the brick explodes and the ball bounces.
When the ball hits a border, the ball bounces.
When the ball hits the bottom border, the ball explodes.

When the ball explodes, the game is over.
When all the bricks are gone, you win
```

## 3.2 Implementation

This section lists and explains the methods used in this project. Some frameworks and libraries that were evaluated were Grammatical Framework (GF) [25] and Nearley [26], two different tools for parsing controlled Natural Languages, and different JavaScript game engines.

Regarding programming methods, a divide and conquer strategy was adopted; the problems were divided and a responsible team member was assigned to each task. In addition, agile development [27] was used, with the team using an iterative approach to programming, meaning the solutions were revised and changed as the development process progressed. A part of the agile process is testing, which was done continuously during development. Another aspect was that the product was developed in increments, meaning a simplified implementation was done initially, and then became more complex as the project moved forward. As previously mentioned, one method of evaluation that was employed was user stories. These were then continuously used to evaluate if the application fulfilled the requirements. Regarding code development, the version-control system git was employed in order to share and sync each team member’s work.

# 4 Design and Implementation

In this chapter, the design and implementation of the different parts of our product is described. This includes the Parser, the Evaluator, the Game Engine, and the User Interface.

Some of the features presented in this chapter are designed and documented but not implemented in the finalised product. This is documented in section 4.11 *Implementation status*, at the end of this chapter.

## 4.1 Design overview

All the information processing done by the application is performed on the client. Here are some advantages of this approach:

- The application remains responsive even on low bandwidth connections and can be used off-line
- Less complex code-base (no need for a communication layer)
- The application can be hosted on any web hotel

In order to speed up the development, an external parser generator, Nearley.js[1], and game engine, Crafty.js[2].

In this thesis we will, when relevant, maintain two levels of description:

- The conceptual level, where we describe the essential design and its underlying principles
- The actual implementation

This division will make the description clearer, we believe, since it gives us the opportunity to present a clear and readily understandable conceptual model. Then, with this model in mind, the actual implementation can be divided in two parts: One that conforms to the design intention, and one that is the result of compromises due to practical considerations.

[1]http://nearley.js.org/
[2]http://craftyjs.com/

The application is implemented using JavaScript, and is intended to run in modern web-browsers that support at least JavaScript ES2015 (often referred to as ES6). The JavaScript language was initially designed for writing small scripts on web pages, but as more and more of modern software is executed directly in the browser, JavaScript has evolved into a language with many powerful features, such as a module system, that makes large applications feasible. However, it remains the responsibility of the developer to choose an appropriate model for the specific application and then to obey the conventions chosen. Over the following we will describe the conventions that we have chosen to use.

Our application treats Natural Language as a rich interface for specifying games. As more and more features of the input language is supported, more grammar rules will be added, with their accompanying semantic functions in the Evaluator. The game engine will also grow as more game objects and possible interactions are added. Our approach to supporting extendability is to let the core structure of the code follow from the data structures used.

## 4.2 Core Structure

Our approach to keeping the code structured and modular depends on two deliberate choices. The first one is to work with a data-structure that fits the domain well. The second is locate all of the evaluator functions in a separate data structure.

The Natural Language processing performed by our application can be seen as a chain of functions that takes pure text, builds a syntax tree, and evaluates the tree into a game representation. The game representation is then interpreted as initialisation code for a game, defining game objects, configurations and rules that maps events to actions.

This could be called an "information processing chain" and in many cases it would be enough to just apply the functions in turn. However, in our app there is a need to accumulate and keep the different stages in order to:

- Produce and show a decorated version of the original Natural Language input.
- Let errors that occurred during processing blame the offending part of the input.

Therefore we have chosen to reify the data of the information processing chain by representing it as an explicit data structure, that we can call "L" for "Language" or "Linguistic". L itself is just a plain JavaScript object, used as a map, where the key/value-pairs of the map represents stages in the processing chain. In order to make L as useful as possible, we have followed some rules for our usage of L. The rules that we describe is closely related to the principles for map types described in the Clojure specification framework `Clojure spec`[3].

[3]`https://clojure.org/about/spec`

L is seen as a relation between components, where the components are the key/value-pairs of the map. The components are separately defined so that they are reusable in other relations The meaning of the relation is constant and does not depend on its *information completeness* at any given time. (*Information completeness* means how many components or key/value-pairs that are present. If a component of L is not known, this must be represented by the total omission of the whole key/value-pair, never by using null as a value of a key.

From now on, a map data type seen in this way and used in accordance with the above restrictions will be referred to as a component-relation map (CR-map).

### 4.2.1 Declarative usage of CR-maps

"Declarative usage" refers to any usage that can take place without the passing of *conceptual time*. Conceptual time is what needs to pass in order for *change* to occur in any of the concepts that is of interest in a given level of description.

CR-map data types have interesting properties: They have the same conceptual meaning, no matter how instantiated their arguments/components are. Also, they can go from a less instantiated state (fewer components present) to a more instantiated state (more components present) without the passing of conceptual time. This means that instantiation is not considered to be "change", or a *side-effect*, but a natural part of a declarative program. Interestingly, under the discipline described above, this still holds for our usage of CR-map types in JavaScript.

In practice this means that the whole of the language processing can be seen as declarative. Only when the processing starts anew, and L is replaced with a new empty L, does conceptual time pass.

### 4.2.2 Functions over CR-maps

It would be advantageous if the implementation of the functions of an information processing chain, such as `parse` or `evaluate`, could stay the same when they are adopted to work on components of CR-maps instead of the raw values. We have chosen to support this by providing a *driver* that accepts information about the input and output components and lifts the original functions into functions over their corresponding CR-map. The lifted functions can be labelled "auto-asscoicative" functions, since they take a partially instantiated relation and gives it back in a more instantiated form.

### 4.2.3 Syntax for declaring CR-map types and function signatures

In order to be able to mention the shape of CR-maps and functions over them in comments and documentation, we have devised an informal notation for component relations, “shapes” or information completeness signatures of relations and auto-associative functions. The notion “shape” is used to specifically refer to what components are demanded in a given context. Auto-associative refers to a function that recreates a whole structure from a part of it. Language processing can be seen this way and this is made explicit by the use of CR-maps in our implementation.

A **relation** with name “R” that can have the components a,b,c:

```
R :: (@a @b @c)
```

The **shape** (information needed in a given context):

```
[@b @c]
```

The shape expression `[@b @c]` means that the components `@b` and `@c` are needed as arguments to a function that will need them.

A **auto-associative function** f that depends on the existence of the component `a`, and contributes the component `b` is written as:

```
f  :: [@a] (+ @b)
```

Assuming the components `a` and `b` have values with types `A` and `B` respectively, the function type of the non-lifted function looks like this:

```
f’  :: A -> B
```

## 4.3 Decomposition of language processing functionality

The language processing of our application takes a text, *parses* it into a syntax tree that is then *evaluated* into a representation of a game. This representation is then simulated using a game engine.

```
L :: (@text @syntax @model @domain)

parse    :: [L @text] (+ @syntax)
evaluate :: [L @syntax] (+ @model)
simulate :: [L @model] (+ @domain)
```

The (@model) component holds the game representation, which is a collection of game objects, configurations, such as “The user controls the paddle” and a set of rules that map events to actions.

Since we use an external game engine (Crafty.js), the game representation is interpreted as initialisation code for Crafty.js. This means that only `parse` and `evaluate` are applicable to our actual code, and `simulate` is not part of our own implementation, just as there is no explicit `@domain` data-structure. The functionality is still present, but the processing takes place inside the game engine.

The above corresponds to the following more familiar types, if we ignore the need to keep the full context.

```
parse     :: String -> Syntax
evaluate  :: Syntax -> Model
```

## 4.4 Grammar/Evaluator Separation

We use a third-party parser generator, Nearley.js, that takes a context-free grammar and produces a general parser using the Earley algorithm[28]. It uses a custom file-format that is compiled into an executable JavaScript file, that can be included in a given code-base. The custom file-format offers the advantage of a succinct and familiar format for defining grammars. But this also means that the post-processing of the recognised syntactic structures must be encoded into the special file-format using a special syntax. This is potentially problematic since the grammar in itself already consists of many rules, and adding a substantial amount of post-processing code will typically reduce readability.

We solved this problem by factoring our all the post-processing code into a separate data-structure. Here follows a description of the strategy used.

### 4.4.1 Tagging the production rules of the grammar

For every production rule in the grammar, we use a function called `tag`. This function does exactly two things: It selects the components of the match result from a clause and tags these components with a label or multiple labels. The following is a simplified example:

```
# definite noun phrase (NP)
Decl -> "there" "are" NP_Indef
    {% tag({ template: 1 })("Assertion","Object") %}

NP_Def -> "the" Noun
    {% tag({ noun: 1 })("NP","Definite") %}

NP_Indef -> "a" Noun
    {% tag({ noun: 1 })("NP","Indefinite") %}
```

The numbers used in the role-object given to `tag` are indices into the array that Nearley.js returns for each clause that matches. The result from running the grammar with the tagging post-processors is an Abstract Syntax Tree: The tag-functions have selected only the important information and tagged them with labels that explain what they are.

It is worth noting that the resulting Abstract Syntax Tree is a data structure with essentially the same recursive structure as the grammar itself.

### 4.4.2 Defining the meaning of the tags

We have chosen to put all the definitions of what the tags mean into one data structure, called `Evaluator`. This data structure is essentially the evaluator function, but it is not executable JavaScript code. This is the work of the *driver*, `evaluate`.

The driver `evaluate` performs the job of traversing the AST, using the labels to look up the applicable function body, and then applying it to the results of recursively invoking itself on each argument. One way to see the work of `evaluate`, is that it recreates the recursive call stack that is executed during parsing.

This is how Evaluate would look for the two rules above. The environment is given to each function as an optional second argument, and the lexicon as an optional third argument.

```
const Evaluator = {
    NP:{
        Definite: ( { noun }, env ) =>
            lookup( env, noun ),
        Indefinite: ( { noun }, env, lex ) =>
            lex[ noun ],
    },
    Assertion: {
        Object: ( { template } , env )
            => assert( env, template ),
        Rule: ...
    },
}
```

## 4.5 Supporting visual feedback for Natural Language

Adequate visual feedback can enhance the user experience by communicating hints to the user about how the system interprets the input text.

Syntax highlighting have become ubiquitous in editors for programming languages. But in most systems that support Natural Language input, (e.g. GameChangineer and WordsEye) it is not supported. This is arguably not ideal; very few programmers can imagine programming with syntax highlighting switched off. It is reasonable to assume that the following properties of visual feedback can contribute to its popularity among programmers:

- Contributes to the programmers intuitive understanding of how the system interprets the input text.
- Gives immediate feedback on whether the input is well-formed or not and increases user confidence.

Given the advantages listed above, we have included support for a limited version of visual feedback. We call it *Code Decoration*, since we have adopted it to the current use case: Traditional syntax highlighting adds colour and style to the different parts of the source code. This is reasonable since programming languages already have plenty of tokens that are used to mark groupings: **<. . . >**, **(. . . )**, [. . . ], **{. . . }** etc. Since they are uncommon in Natural Language, we have adopted them as part of the text-decoration, with the intention of providing the user with readily accessible information on the interpreted groupings and semantics of the text.

The implementation of text decoration in our application is straight-forward, due to the fact that we already produce a fully tagged syntax tree. All that is needed is to add an extra component `@deco` to the relation `L`. Since `L` is a CR-map type, no existing code will be affected, since an additional component can never take away information and never effect the shape declarations of code that do not use the extra information. The content of the new component is computed inside the `tag` function, due to the fact that the concrete part of the syntax otherwise is discarded there.

Here follows an example of a decorated indefinite noun phrase. The decorated form that appear as part of the visual feedback is:

```
<a ball>
```

Based on the tags and the concrete tokens of the syntax, a HTML-representation of the code is computed. Here is a HTML fragment corresponding to the above decorated code.

```
<span class="NP␣Indefinite">a ball</span>
```

All the visual appearance of the decorated code, including added brackets, are defined using CSS (Cascading Style Sheets)[4]. Here follows the CSS that completes the example above, telling the browser to render an indefinite noun-phrase surrounded by grey bold angle-brackets.

```
span.NP.Indefinite::before {
    content: "<";
    color: #888;
    font-weight: bold;
}
span.NP.Indefinite::after {
    content: ">";
    color: #888;
```

[4]https://www.w3.org/Style/CSS/

```
        font-weight: bold;
    }
```

We have chosen to render all characters that are part of the syntax decoration in bold, grey style in order to facilitate for the user to see what is decoration and what is the actual characters of the input. Otherwise, we have chosen to colour and style the Natural Language code in a way similar to traditional syntax highlighting.

Since code decoration for Natural Language is not wide-spread, the question of what mark-up to use is open, and experience might lead in one or another direction. We take the stance that it is likely that it can enhance the user experience, if done tastefully; it is straightforward to support and easy to customise using CSS.

## 4.6 Supporting localised error indication

When trying to program a game, but something does not work, it is useful to get precise information about what is wrong and where the error is located. Therefore we have chosen to make localised error messages an integral and central part of the design.

An error in code could be just a simple mistake, or it could be a sign of an inconsistency between what the user thinks that the system means with an expression and what the system actually means. What the system can do is to provide the user with information about what is wrong and where the error is located.

The application has three subsystems that can detect errors: The Parser, the Evaluator and the Game Engine. For the parser and the game engine we use external libraries, so the availability of error type and localisation information depends on these libraries.

For the evaluator, we have designed error handling as part of our core data-type; the CR map type.

When an error occurs, the component that was supposed to be produced is not available. According to our rules for CR-map types, this component must be left out entirely. What is available is instead the following information:

- The component we were trying to compute when the error occurred
- The error message
- The path in the AST to where the error was detected

The error information is encoded in the L relation as a component `(@error)`. The value of the error component is a CR-map:

```
        (@error: (@goal @path @message))
```

Since a relation tagged with an error takes precedence over non-error semantics, the code handling errors is located in the evaluate function itself. This means that if the result of evaluating an argument is tagged with `@error`, the current operation is not performed. Instead, the error found is annotated with correct path information to the level currently evaluated and then returned. When the error reaches the top level the UI enters an error state, showing the user the error, with the error message available for inspection.

## 4.7 Ambiguity

While disambiguation using statistic and/or semantic information is an interesting topic in itself, it is a possible source of uncertainty: Even if the user is 100% sure of the meaning of a syntactically ambiguous expression, they might not be sure about how the system interprets it. Since the goal is to offer the user an opportunity to develop Computational Thinking, it would be preferable to make the system as transparent as possible.

To keep the grammar from being ambiguous, restrictions are used. For example, preposition phrases that are applicable both to nouns and verbs are not used. This approach will not be feasible if the application is extended to handle more games and more varied input. A solution to this would be to give the user the ability to format their input text, for example by using indentation, in a way that would make attachment explicit. Such input is currently not supported.

## 4.8 Supporting domain vocabulary: The Colour Domain

To master a Natural Language means, among other things, to have an ability to refer to and describe anything in the world, by using general concepts that can be refined if needed. An example of this principle is the colour domain: Using only a handful of general color words and a few modifiers, a human user can refer to any color in general terms, and if needed, refine descriptions to within a desired precision.

We have chosen to offer basic support for colour vocabulary with a sufficient degree of cognitive realism for the vocabulary to be useful in practice. In the sections that follow we describe the principles used.

### 4.8.1 Supported colour vocabulary

We have chosen to provide coverage for the following vocabulary:

**Basic colour words**
*black, grey, white, red, orange, yellow, green, blue, violet*

**Basic colour modifiers**
*light, dark, strong, weak*

**Basic modifiers of modifier**
*very, somewhat*

With these 14 words we offer basic coverage of the colour domain. Here is our model:

We use the *hue – saturation – lightness* (HSL) encoding, since it offers a decent mapping to Natural Language concepts such light and dark, strong and weak. This is an advantage compared to other representations such as the RGB colour model, where no such straight-forward mapping is available.

When a colour word is used in English, the word class is far from obvious[29]. We support only two possible cases: adjective usage and proper noun usage (see example below). We also communicate to the user which grammatical class the system has attributed to a color word, by decorating the proper noun form with title case. E.g.

```
(1) a green ball
(2) a ball with colour Green
```

Where (1) is an adjective usage and (2) is a proper noun usage.

Here is an example of the concrete representation of representation of the color green:

```
@key: "color",
@value:
    @hue: 125,
    @saturation: 65,
    @lightness: 50,
@weight:1,
```

When used as an adjective, the colour information refines an already existing colour component of an entity model. When used as a proper noun, only the `@value` component is used, and the `@key` component is provided separately as a noun phrase (2) above, and only the `@value` is used from the colour word.

## 4.8.2 Colour modifiers

We have chosen a *weighted average* model of modification that is straight-forward to implement, and gives a decent distribution of colour models as a result of combining colour adjectives with modifiers. A tentatively more realistic model that we considered was to support intervals, and then use strategies such as “typical” or “random” when a coloured object is instantiated and there is need for a concrete colour.

The model is implemented as follows: A colour modifier such as *light* has an internal ideal value, an *attractor* and a weight. When applied to a entity model in the colour domain, the corresponding property of the entity model is attracted in the direction of the attractor. Here is the model for **light** in usages such as “light green”:

```
@key: "color",
@value:
    @lightness: 0,
@weight:1,
```

Now modifier-modifiers such as *very* and *somewhat* can be supported by letting them modify the `@weight` component of a modifier.

The result of evaluating an utterance such as “very light green” would then be:

```
@key: "color",
@value:
    @hue: 125,
    @saturation: 65,
    @lightness: (50*1 + 100*2)/(1+2),
@weight:1,
```

## 4.9 Game engine

The mapping from the game representation to objects in the game engine is fairly simple. If parsing and evaluation succeeds, the Evaluator will return a JavaScript object containing two lists of objects: One of entities to be created, and one of rules. Each entity object contains a key-value map of values that indicates the type, look and behaviour of the entity. The entities are initiated empty, and components are then added iterative based on the content in the entity object. The game engine used in this project, CraftyJS, uses the ECS pattern, as discussed in section 2.3 *Entity Component System*. This makes it trivial to create a game object and dynamically adding behaviour this way.

In the second list of rules, each rule object contains an event, an action triggered by the event, and a subject of the event.The rules are added locally to each object affected by a rule. For example, if a rules says a ball should explode when hitting a wall, each ball entity in the game representation will get the rule added to them. Another solution to this would to have the rules available globally, and not connected directly to an entity. This could have made the rules more flexible to change and sped up the instantiation of the game representation. However, it would require the development of a module to handle rules and events, which was too time consuming for the scope of this project.

Collision detection was handled for us by CraftyJS, however what happens after a collision had to be programmed by ourselves. CraftyJS uses Separating Axis

Theorem (SAT) [30] to detect collisions which returns a minimum translation vector (MTV) [31], that is the shortest distance the object has to be moved in a certain direction for the objects to no longer intersect. For the instance of bouncing objects, this vector was used as a reflection line, so that the object's old velocity vector was mirrored in relation to the line, giving us the object's new velocity vector. This was the formula used:

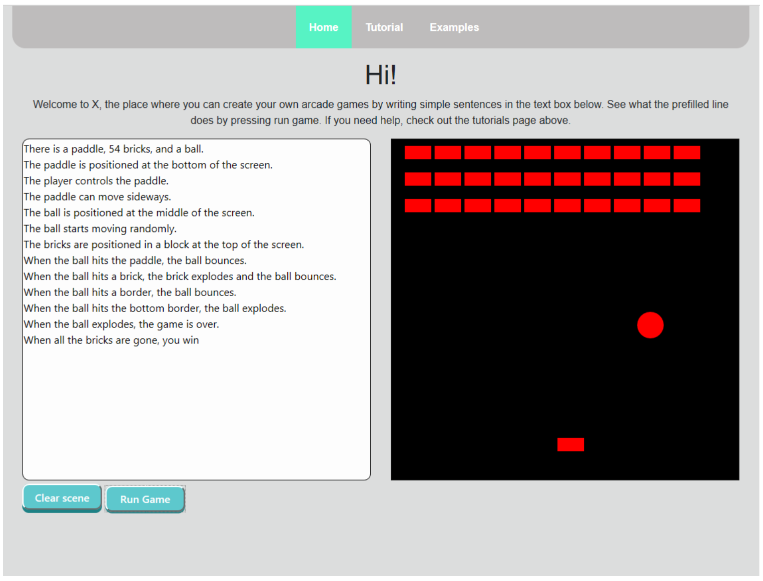
$$V_{new} = V_{old} - 2(V_{old} \cdot \hat{n})\hat{n}$$

*where n is normal vector of the MTV*

## 4.10 User interface

Designing and developing a well thought out and responsive User Interface (UI) is an important part of making the application an attractive and effective programming education tool. The UI for this application was built using React, a JavaScript framework. Since this application is aimed at children aged 10-18 that are new to programming the idea was to make the UI as self explanatory as possible, so as to lower the overhead of learning how it worked and instead enabling the user to start testing directly.

We chose to make a simple UI, see Figure 3. The focus on the main page is the input field and game view, since we did not want users to have to see a long introduction text every time they went to our application. As a new user you might want a tutorial or examples. Therefore there are tabs for those easily available at the top of the page.



Hi!

Welcome to X, the place where you can create your own arcade games by writing simple sentences in the text box below. See what the prefilled line does by pressing run game. If you need help, check out the tutorials page above.

```
There is a paddle, 54 bricks, and a ball.
The paddle is positioned at the bottom of the screen.
The player controls the paddle.
The paddle can move sideways.
The ball is positioned at the middle of the screen.
The ball starts moving randomly.
The bricks are positioned in a block at the top of the screen.
When the ball hits the paddle, the ball bounces.
When the ball hits a brick, the brick explodes and the ball bounces.
When the ball hits a border, the ball bounces.
When the ball hits the bottom border, the ball explodes.
When the ball explodes, the game is over.
When all the bricks are gone, you win
```

**Figure 3:** A screenshot of what the UI looks like, featuring the Brick Breaker example given in subsection 3.1.2 *Test cases*. The page is split in the middle with the left side having the input field, once you click 'Run Game' it starts playing on the right side.

Bootstrap, a framework for CSS, was used to provide styling for the components. Bootstrap also enables the development of responsive application s with its grid system. All the content is organised on a grid with 12 columns, which can be grouped in any matter.

## 4.11 Implementation status

Due to time constraints, not all of the planned and designed functionality could be implemented in the final version of the product. Below is the status of the different parts of our product at the end of the project.

**Grammar for the intended scope**
The Grammar currently has coverage that is limited to the fundamental needs of one arcade game, *Brick Breaker*.

**Code Decoration**
Supported – a decorated version of the input text is shown, but is not integrated into the text input field.

**Error Indication**
Not supported, errors are not indicated as part of the decorated code.

**Colour Domain Vocabulary**
Colours can be referred to using basic colour words and modifiers as specified. Only the adjective form and not the proper noun form of basic color words is supported.

**Progress Indication**
Not supported

# 5
# Discussion

This chapter will describe what impressions and insights the project resulted in. It will also discuss what work this could lead to, and how the project could be further developed in the future.

## 5.1 Product

As mentioned in section 4.11 *Implementation status*, we did not implement as much of our design as planned. For example, error indication was a central concept of our project, but was not be implemented. Several features that were designed but not implemented are fairly straightforward and could potentially contribute positively to the user experience of the app, see more under Future Work.

## 5.2 Contribution to Computational Thinking

Our conviction is that our application does contribute to learning Computational Thinking, however perhaps not to the extent we initially thought. This could be due to the fact that the coverage of what is possible to express is limited, i.e how much Natural Language the parser can interpret as concepts that are supported by the game engine.

- *Formulating problems in a way that enables us to use a computer and other tools to help solve them.*

  In order to use our product to create a game, the user will first have to formulate what the problem is. Once this is done the user can implement a solution, which has to be structured in a way that a computer, or in this case the parser, can interpret.

- *Automating solutions through algorithmic thinking (a series of ordered steps)*

  Our product accepts a small subset of Natural Language, and the user will be programming and writing small algorithms in that language. This promotes algorithmic thinking.

- *Identifying, analysing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources*

  We believe our product could contribute to this skill if a concept discussed in future work is implemented: challenges. By implementing different challenges for the user to solve a task - such as producing a game in as few lines as possible - our product could teach a user how to analyse and implement solutions that are more effective and efficient.

- *Generalising and transferring this problem solving process to a wide variety of problems*

  Since our product should be versatile and able to produce a wide array of games, the users should be able to use and learn from their past solutions to solve future challenges.

## 5.3 Future work

In the following section we will discuss how the application could be extended.

### 5.3.1 Error messages

The use of an external parsing library proved to be time-saving, but lead to imprecise error messages for grammatical errors. Currently only a position in the text is shown. A natural goal would be to support partial parsing, where everything that is correct is still parsed and visualised with normal code decoration, and what is not correct is highlighted. A way to support this would be to build a custom dual strategy parser, or alternatively, to access the parse forest of the existing third-party parser generator. An advantage of a fully integrated parser would be that it would be possible to mention it at the Natural Language level itself, making it possible to say things like `the result of parsing the string "a big read ball"`. Giving the user the ability to express everything within the system with a clear meaning this way could arguably have some pedagogical value.

### 5.3.2 Multiple source languages

In it's current version, the application only supports one input language: English. An interesting extension would be to support several source languages. This could open up the possibility of more people using the application, and in the language they are most comfortable with. Allowing the user to write in their first language will strengthen the benefits of using Natural Language in the first place.

### 5.3.3 User interface

One of the things that could be done to make the UI more communicative is to indicate how far in the process the user has come when an error occurs. The different steps could be syntactic, indicating that the input is well formed according to the rules, another could be semantically meaning that the users intention is properly expressed. This could be done using a progress bar, which shows if the which steps of the processing that was completed.

Another thing that could be done to improve the UI so that it better supports the user's learning is more extensive examples and tutorials. While learning by doing is encouraged, some explanation on how our controlled Natural Language is structured would make the application easier to use.

### 5.3.4 Challenges

Adding challenges to the product could be a good way to help users improve their skills. Examples of challenges are how to write a game in as few lines as possible, or how to write a game first in a very forgiving controlled Natural Language, and then a more constrained version.

## 5.4 Evaluating the application

There are several possible strategies for evaluating an application. The following sections will discuss such evaluation strategies that would have been appropriate for our project.

### 5.4.1 Survey

A potential user group of the end product will be students. There are several methods of evaluating how well the product achieves its purpose involving the students. One of these is to survey the students before and after using the application.

Teachers are another group of users that it would be interesting to survey what their opinion of the application themselves and the assess their opinions. This can be done by allowing the teachers to use the application for themselves or in their teachings, and then survey the teacher on how well it worked in a education environment.

### 5.4.2 A-B testing

Using a test in logical thinking, which would asses how proficient the students are in logical thinking before and after spending time with the application, on could preform what is called A-B testing [32]. This is done by letting one group of students use the application, but not the other, then allowing both groups to take the test. The results of the two groups are then compared and studied. In the event that there are no students available, one could imagine using university students in this course.

### 5.4.3 User analysis

Another possible way to measure the results of the application is to track the user, and analyse the data. For instance, it would be interesting to see how often the user commits a mistake, and if this is changes over time. Furthermore, one could analyse the games the user creates, and see if they become more complex, and how this relates to the other aspects analysed, especially over time.

# 6
# Conclusion

In conclusion, the application is modular; it is possible to change the implementation of all the parts, as long as the interfaces between them are followed. The application is also extendable; the current implementation does not need to be changed in order to add more functionality.

We conclude that, according to the description of Computational Thinking given in section 2.1, our project contribute to the following aspects of teaching Computational Thinking:

- Formulating problems in a way that enables us to use a computer and other tools to help solve them.
- Automating solutions through algorithmic thinking (a series of ordered steps)
- Identifying, analysing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources
- Generalising and transferring this problem solving process to a wide variety of problems

# Bibliography

[1] J. A. Van Dijk, *The deepening divide: Inequality in the information society.* Sage Publications, 2005.

[2] Y. B. Kafai and Q. Burke, “Computer programming goes back to school”, *Phi Delta Kappan*, vol. 95, no. 1, pp. 61–65, 2013.

[3] A. Balanskat and K. Engelhardt, “Computing our future: Computer programming and coding - priorities, school curricula and initiatives across europe”, Tech. Rep., Oct. 2015. [Online]. Available: `https://www.researchgate.net/publication/284139559_Computing_our_future_Computer_programming_and_coding_-_Priorities_school_curricula_and_initiatives_across_Europe`.

[4] M. LLC. (). Förändringar och digital kompetens i styrdokument, [Online]. Available: `https://www.skolverket.se/temasidor/digitalisering/digital-kompetens` (visited on 02/06/2019).

[5] (2017). Gamechangineering: Logic, language, game design | ece | virginia tech, [Online]. Available: `https://ece.vt.edu/news/article/gamechangineer` (visited on 02/13/2019).

[6] S. Papert, *Mindstorms: Children, computers, and powerful ideas.* New York, NY, USA: Basic Books, Inc., 1980, ISBN: 0-465-04627-4. [Online]. Available: `http://worrydream.com/refs/Papert%20-%20Mindstorms%201st%20ed.pdf`.

[7] A. Goldberg and D. Robson, *Smalltalk-80: The language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[8] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, “The scratch programming language and environment”, *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.

[9] B. Coyne and R. Sproat, “Wordseye: An automatic text-to-scene conversion system”, in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’01, New York, NY, USA: ACM, 2001, pp. 487–496, ISBN: 1-58113-374-X. DOI: `10.1145/383259.383316`. [Online]. Available: `http://doi.acm.org/10.1145/383259.383316`.

[10] M. Tedre and P. J. Denning, “The long quest for computational thinking”, in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling ’16, Koli, Finland: ACM, 2016, pp. 120–129, ISBN: 978-1-4503-4770-9. DOI: `10.1145/2999541.2999542`. [Online]. Available: `http://doi.acm.org/10.1145/2999541.2999542`.

[11] J. Wing, “Computational thinking”, *Communications of the ACM*, vol. 49, pp. 33–35, Mar. 2006. DOI: `10.1145/1118178.1118215`.

[12] ——, “Research notebook: Computational thinking—what and why”, *The Link Magazine*, pp. 20–23, 2011. [Online]. Available: `http://people.cs.vt.edu/~kafura/CS6604/Papers/CT-What-And-Why.pdf`.

[13] S. Grover and R. Pea, “Computational thinking in k–12”, *Educational Researcher*, vol. 42, no. 1, pp. 38–43, Jan. 2013. DOI: `10.3102/0013189x12463051`.

[14] J. Good, A. Yadav, and P. Mishra, “Computational thinking in computer science classrooms: Viewpoints from cs educators”, in *Proceedings of Society for Information Technology & Teacher Education International Conference 2017*, P. Resta and S. Smith, Eds., Austin, TX, United States: Association for the Advancement of Computing in Education (AACE), Mar. 2017, pp. 51–59. [Online]. Available: `https://www.learntechlib.org/p/177274`.

[15] V. Barr and C. Stephenson, “Bringing computational thinking to k-12: What is involved and what is the role of the computer science education community?”, *ACM Inroads*, vol. 2, pp. 48–54, Mar. 2011. DOI: `10.1145/1929887.1929905`.

[16] I. S. for Technology in Education (ISTE) & Computer Science Teachers Association (CSTA), *Operational definition of computational thinking for k–12 education*, 2011. [Online]. Available: `https://id.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf?sfvrsn=2`.

[17] S. Papadakis, M. Kalogiannakis, and N. Zaranis, “Developing fundamental programming concepts and computational thinking with scratchjr in preschool education: A case study”, *International Journal of Mobile Learning and Organisation*, vol. 10, pp. 187–202, Jul. 2016. DOI: `10.1504/IJMLO.2016.077867`.

[18] E. F. Anderson, S. Engel, L. McLoughlin, and P. Comninos, “The case for research in game engine architecture.”, 2008. [Online]. Available: `http://eprints.bournemouth.ac.uk/24322/1/FP8GEA.pdf`.

[19] A. Thorn, *Game engine design and implementation.* Jones & Bartlett Publishers, 2011.

[20] D. M. Hall, “Ecs game engine design”, 2014. [Online]. Available: `https://digitalcommons.calpoly.edu/cpesp/135/`.

[21] B. J. Cox, “Object-oriented programming: An evolutionary approach”, 1986.

[22] D. Wiebusch and M. E. Latoschik, “Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems”, in *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, IEEE, 2015, pp. 25–32.

[23] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Automata theory, languages, and computation*. Pearsn Education Limited, 2008, ISBN: 1-292-03905-1.

[24] M. Cohn, *User stories applied: For agile software development*, ser. Addison-Wesley signature series. Addison-Wesley, 2004, ISBN: 9780321205681. [Online]. Available: `https://books.google.se/books?id=SvIwuX4SVigC`.

[25] *Grammatical framework*. [Online]. Available: `https://www.grammaticalframework.org/`.

[26] Kach, *Kach/nearley*, Feb. 2019. [Online]. Available: `https://github.com/kach/nearley`.

[27] M. Alexander, *Agile project management: A comprehensive guide*, Jun. 2018. [Online]. Available: `https://www.cio.com/article/3156998/agile-development/agile-project-management-a-beginners-guide.html`.

[28] J. Earley, “An efficient context-free parsing algorithm”, *Commun. ACM*, vol. 13, no. 2, pp. 94–102, Feb. 1970, ISSN: 0001-0782. DOI: `10.1145/362007.362035`. [Online]. Available: `http://doi.acm.org/10.1145/362007.362035`.

[29] R. Langacker and R. Langacker, “Cognitive grammar: A basic introduction”, in. Oxford University Press, USA, 2008, p. 102, ISBN: 9780195331950. [Online]. Available: `https://books.google.pn/books?id=6QsSDAAAQBAJ`.

[30] K. S. Chong, *Collision detection using the separating axis theorem*, 2012. [Online]. Available: `https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169`.

[31] Crafty. (). Collision, [Online]. Available: `http://craftyjs.com/api/Collision.html`.

[32] J. Gregory and L. Crispin, *More agile testing: Learning journeys for the whole team*, ser. Addison-Wesley Signature Series (Cohn). Pearson Education, 2014, pp. 203–204, ISBN: 9780133749564. [Online]. Available: `https://books.google.se/books?id=uq-pBAAAQBAJ`.