Thapl—A Theatrical Programming Language

Matan I. Peled
Thapl—A Theatrical Programming Language

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Matan I. Peled

Submitted to the Senate of the Technion — Israel Institute of Technology
Heshvan 5780 Haifa November 2019
This work was carried out under the joint supervision of Prof. Yossi Gil and Prof. David H. Lorenz, in the Faculty of Computer Science.

Some results in this thesis have been published as articles or talks by the author and research collaborators in conferences and journals during the course of the author’s magisteral research period, the most up-to-date versions of which being:


Acknowledgements

To my parents Dafna and Michael, I am grateful for your endless love and support throughout the years, I literally wouldn’t have been here without you.

I am indebted to my advisors, Prof. Yossi Gil and Prof. David H. Lorentz, for patiently guiding me through tight deadlines, and for teaching me what research actually is.

A most heartfelt thanks goes to my friends for keeping me sane, to Nitzan for making sure I go swimming at least once a week no matter what, and to all those who helped me during my research; especially Shiri and Hila who helped improve this document. Finally, I’d like to thank Ira for being patient and understanding, for putting up with me when my stress levels went through the roof, and just for being awesome.

The generous support of the Technion is acknowledged. This research was supported in part by the Israel Science Foundation (ISF) under Grant No. 1440/14.
# Contents

List of Listings

List of Figures

List of Terms

Abstract

1 Introduction
1.1 The Problem ........................................... 3
1.2 Shortcomings of Current Solutions ......................... 4
1.3 Thesis .................................................. 4
1.4 A Simple Example in THAPL ............................... 5

2 Background
2.1 Presentation Software ................................... 9
2.2 Visualization Languages ................................ 13

3 Tutorial by Example
3.1 Introduction ............................................ 18
3.1.1 Defining characters ................................ 18
3.1.2 Defining actors ..................................... 19
3.2 Action ................................................... 19
3.2.1 Defining verbs ...................................... 21
3.2.2 Controlling time .................................... 21
3.3 Conclusion ............................................. 21

4 Syntax
4.1 Lexical Matters .......................................... 26
4.1.1 Literals ............................................. 27
4.1.2 Tokenization sequence .............................. 29
4.2 Program Structure ....................................... 30
4.3 Declarative Definitions .................................. 32
4.3.1 Type System and Primitive Types ................. 34
## 4.4 Directives: Sentences, Subjects, Verbs, and Modifiers ........................................... 35
  4.4.1 Compound Directives ............................................................... 35
  4.4.2 Atomic directives ................................................................. 36

## 4.5 Practicalities and Ambiguities .................................................................................. 38
  4.5.1 Parsing ......................................................................................... 38
  4.5.2 Ambiguities .................................................................................. 39

## 5 Semantics: Language and Animation ......................................................................... 41
  5.1 Object Model ..................................................................................... 41
    5.1.1 Object Creation ........................................................................... 42
    5.1.2 Inheritance ................................................................................. 42
  5.2 Name Resolution in THAPL ................................................................................. 42
    5.2.1 Implementation Using Context Data-structure ................................ 43
    5.2.2 Simulating Other Languages ....................................................... 46
  5.3 Animation Semantics ......................................................................................... 47
    5.3.1 Primitives .................................................................................... 47
    5.3.2 Solving for Slides ......................................................................... 48

## 6 Physical Constraint Model: Springs .......................................................................... 51
  6.1 The Physical Model ......................................................................................... 53
    6.1.1 Linear springs .............................................................................. 53
    6.1.2 The spring group(s) ...................................................................... 56
    6.1.3 Discrete non-linear springs ............................................................. 58

## 7 Results ..................................................................................................................... 61
  7.1 Fitness For Purpose ......................................................................................... 61
    7.1.1 Complex Animations ..................................................................... 61
    7.1.2 Simulating Logic Circuits ............................................................... 62
    7.1.3 Hannukah Menorah ....................................................................... 63
  7.2 Flexibility ........................................................................................................... 65
  7.3 Analysis .............................................................................................................. 66

## 8 Conclusion and Future Work .................................................................................... 69
  8.1 Related Research ............................................................................................. 70
    8.1.1 Musical Composition ...................................................................... 70
    8.1.2 Name-binding ............................................................................... 71
    8.1.3 Other DSLs .................................................................................... 71
  8.2 Future Directions ............................................................................................... 74
  8.3 Closing Notes ..................................................................................................... 75
# A Code Listings

A.1 THAPL Compiler .............................................. 83
   A.1.1 Grammar ................................................. 83
A.2 LITTLE SMALLTALK ............................................ 86

# B Code Listings for \TeX\ and THAPL Examples

B.1 Towers of Hanoi ........................................... 89
   B.1.1 \LaTeX ................................................. 89
   B.1.2 THAPL ................................................. 92
B.2 Simulating a Logic Circuit ............................. 93
   B.2.1 \LaTeX ................................................. 93
   B.2.2 THAPL ................................................. 95
B.3 Hannukah Menorah ....................................... 97
   B.3.1 THAPL ................................................. 97
B.4 Convolution .............................................. 98
   B.4.1 THAPL ................................................. 98
B.5 Intersection ............................................. 99
   B.5.1 THAPL ................................................. 99

# C Syntax Derivation Rules

101

# D Installation Guide & Quick Start

D.1 Installing and running the main program ............. 105
D.2 Installing the Jupyter extension ...................... 106

Hebrew Abstract .............................................. i
**List of Listings**

1.1 First example of a THAPL program .................................. 4  
1.2 A simple THAPL program animating intersecting circles .......... 6  
2.1 A simple animation in \LaTeX .................................. 10  
2.2 Animation using TikZ’s animation facility. .......................... 12  
2.3 A simple animation in CSS .................................. 13  
2.4 Towers of Hanoi in ANIMALSCRIPT [Rössling and Freisleben, 2001] . . . 14  
2.5 Bubble sort in ANIMALSCRIPT2 [Rößling et al., 2004] ............. 14  
3.1 Empty THAPL Program .................................. 18  
3.2 Empty THAPL Program, with load statement ....................... 18  
3.3 Definition of pole and disc .................................. 18  
3.4 Adding has properties to pole and disc .......................... 19  
3.5 Defining actors for poles and discs .............................. 19  
3.6 Adding a simple action to the animation .......................... 20  
3.7 Defining a can property .................................. 21  
3.8 Solving the Towers of Hanoi .................................. 22  
3.9 Solving the Towers of Hanoi quickly using **MEANWHILE** .......... 23  
4.1 Organization and structure of a THAPL program ..................... 30  
4.2 An example of a simple entity definition. ......................... 34  
5.1 Thapl program demonstrating the object model. .................... 42  
5.2 Illustration of the Context concept using PYTHON syntax ........ 44  
5.3 Context construction for “play grouping” using PYTHON syntax ... 45  
5.4 Context construction for “play grouping” with link to enclosing scope . 45  
5.5 Context construction for “play grouping” with inheritance and scopes 46  
7.1 Stack Exchange answer for complex animation ..................... 62  
7.2 THAPL version of “complex animation” ........................... 62  
7.3 Stack Exchange answer for circuit simulation (excerpt) .......... 63  
7.4 THAPL implementation of a circuit simulation presentation (excerpt) ... 63  
7.5 Hannukah Menorah TikZ animation, from Stack Exchange .......... 64  
7.6 Hannukah Menorah: THAPL Entity definitions (excerpt) .......... 64  
7.7 Hannukah Menorah: THAPL **ACTION** directives (excerpt) ....... 65  
A.1 thapl.lark: Grammar definition for THAPL ....................... 83
A.2 test_smalltalk.py: Demonstration of context implementation of the Smalltalk type hierarchy

B.1 \LaTeX\ Implementation of Towers of Hanoi
B.2 Complete ThAPL implementation of Towers of Hanoi
B.3 \LaTeX\ implementation of a flip-flop digital circuit
B.4 ThAPL implementation of a flip-flop digital circuit
B.5 Complete Hannukah Menorah ThAPL implementation
B.6 Convolution presentation: ThAPL implementation
B.7 Intersection presentation: ThAPL implementation

D.1 Example code to run within Jupyter
D.2 Installation script for ThAPL
List of Figures

1.1 A single frame from the animation shown in Listing 1.2 ............ 7
2.1 TikZ example visualized ........................................ 12
3.1 The Towers of Hanoi, starting position .......................... 17
3.2 Output of the program shown in Listing 3.5 .................... 20
3.3 Truncated output of the program shown in Listing 3.6 .......... 20
5.1 Mutation profile ................................................. 48
6.1 Abstract syntax tree diagram for a compound directive  .......... 52
6.2 Spring system for a compound directive ........................ 52
7.1 Logic Circuit ..................................................... 62
7.2 Hannukah Menorah .............................................. 63
7.3 The first frames from the Hanoi presentation .................... 65
7.4 Four frames from a presentation explaining the convolution operation 66
7.5 Part of a presentation of two moving, intersecting circles ....... 67
List of Terms

actor  In Thapl, concrete stage players that will be rendered in the produced animation, and which exist at runtime. 6, 7, 18, 19, 21, 22, 31–34, 37, 39–43, 74–76

character  In Thapl, abstract archetypes of actors, similar to abstract classes in other languages. 7, 18, 19, 22, 31, 32, 41, 43, 74

Context  The Context data-structure, discussed in Chapter 5, is an extensible dictionary-like mapping used to implement name-binding in Thapl. 5, 8, 32, 39, 41, 43–47, 71

declarative programming paradigm  A programming paradigm that emphasizes the logic of a computational activity without explicitly describing the control flow the computation necessitates. 1, 5, 69, 70, 76

Dramatis Personæ  The listing of characters that is placed before each scene in a theatrical play, letting the reader know who will appear in advance. 5, 6, 38

DSL  Domain-specific language. 5, 9, 10, 13, 69–71

grammar  A formal grammar is a set of production rules that describe how to form documents from the language's alphabet. These rules comprise of terminal symbols, which are the root symbols in the language describing concrete words, and non-terminal symbols, which are expected to be replaced with terminal symbols using a production rule. 25, 36, 38

imperative programming paradigm  A programming paradigm in which a program consists of statements that mutate state, describing how a program achievable a result, often without directly specifying what that result should be. 7, 13, 14, 73

object  Object-oriented programming is an programming paradigm emphasizing objects, which contain both data and code, and modelling computations as interactions between objects. 32, 34, 35, 41, 42, 46, 47, 69, 70, 76
**programming paradigm** A way to classify programming languages by their features, or the style of programming which they encourage. A programming language may be categorized into more than one paradigm. 70

**proto-token** A proto-token is a temporary artifact used in the lexing of THAPL, representing a lexical unit that is a part of a token. 29

**spring** An elastic mechanical device that applies a force when compressed or stretched. In THAPL, springs are used to calculate the duration of animations. 8, 37, 38, 51–59, 69, 75

**token** A token is the atomic unit of parsing. 25, 26

**WYSIWYG** “What you see is what you get” is a type of graphical editing system wherein the content being edited resembles the final product, which may be a printed document, a slideshow, or a web-site. 5

**WYSIWYM** “What you see is what you mean” is an editing software paradigm wherein the structure of the contents is emphasized over the exact way the contents will be rendered in the final product. 5
Abstract

This work explores an innovative approach to the declarative and imperative paradigms of programming languages. To demonstrate this approach, I developed a prototype domain-specific language inspired by the scripts of theatrical plays, hence dubbed THAPL. THAPL’s intended use is in the context of animation generation in “slide-show” presentations. There are two different ways to integrate animations in a presentation. The first is to embed an animation within a single slide, so that the animation starts when that slide starts and stops when continuing to the next slide, as if it were a video. The other is to present the animation as a series of slides, so that each slide is static, like a flip book or a kinetoscope; THAPL embraces this latter approach. The theatrical play metaphor encompasses the concept of a classical theatrical play script (e.g., Shakespearean play), enumerating the “Dramatis Personæ,” scenery, text, and actions (e.g., “exit chased by a bear”). THAPL is practical for real-world uses such as slide-shows for academic courses (especially those that deal with algorithms or graphs), product presentations, and fiscal reviews.

THAPL expands the declarative and imperative paradigms by introducing a declarative vocabulary for describing actors and actions on those actors, as well as a behavior language with specialised constructs for describing sequential and concurrent actions.

Although programming languages that create presentations and animations already exist, they either operate at unreasonably low levels of abstraction which are ill-suited for creating animations, or are aimed at authoring other media types (e.g., films), and only support slide-show creation as an after-thought. Conversely, existing software tools for creating slide-shows are not designed for animations. Modifying an existing animation is hard, and for solutions that utilize binary formats, fundamental software-engineering tools such as source control and diff are impractical.

This work argues for the application of ideas from the declarative programming paradigm in order to focus on specifying behavior, which enables presentation authors to create maintainable presentations. Using THAPL, the programmer describes the animation in a structured way, declaring relations between events. It is then the compiler’s responsibility to sort out the details, e.g., enumerating slides and the duration of each part of the animation. This concept can also be applied to other domains in which the main concern is specifying behavior over time.
Chapter 1

Introduction

A presentation, delivered by a person in front of an audience, is in many respects superior to a written report. A presentation often incorporates visual aids, in the form of a slide-show, and a key differentiator between a passable presentation and an excellent presentation is the quality of the slide-show. A good slide-show augments the speaker, presenting the information that the speaker discusses, and accentuates her points; in that regard, it is often said that “a picture is worth a thousand words”, and it can be surmised that animations are worth even more. Whenever a speaker feels the need to gesticulate in order to explain something, that give a good notion that an animation may be appropriate.

Animations are not merely eye-candy, serving aesthetic purposes but rather serve to illustrate ideas that are difficult to convey with words and static images alone, thus enhancing the viewer’s understanding of the material, and adding clarity to concepts. They may be used to add a time dimension to a presentation, enabling the speaker to describe a process and create a sense of momentum in the story being told. Even the simple technique of adding information to a diagram in a gradual manner helps the audience focus on the one thing that is important in a presentation—the presenter’s point.

1.1 The Problem

The desired effect of adding an animation to a slide-show is, as mentioned above, making the presentation clearer by engaging the audience and demonstrating the main ideas in a visual fashion. Making such presentations with animations, however, is difficult, as drawing them frame-by-frame is tedious, and other approaches still require keeping track of the state of the animation manually.

After the creation of the animated slide-show, the issues continue. Maintaining them, i.e., fixing mistakes and making small changes, is even harder than the initial effort, since the animations were constructed in an ad-hoc manner. This process of iterative enhancements to presentations is relevant in some contexts, especially didactic
ones. Consider a slide-show used for lectures in a university course; the same presentations are repeated every semester, and the lecturer is likely to want to adjust minor details to draw more attention to the point she deems more important, or correct issues that were brought to her attention.

1.2 Shortcomings of Current Solutions

Current tools do not treat animations as a first-class feature, instead concentrating efforts on other aspects of slide-show construction. Although programming languages that create presentations and animations already exist, they either operate at unreasonably low levels of abstraction which are ill-suited for creating animations or are aimed at authoring other media types (e.g., films), and only support slide-show creation as an after-thought.

Most presentations today are created using GUI tools like PowerPoint, Keynote, LibreOffice Impress, or other, equivalent tools. These are fine tools that are fit for purpose, but they do not excel at creating animations. A major problem is that animations built using these tools are difficult to modify; often the easiest option is to start over from scratch. Another limitation is encountered when we desire to have two animations occur on the screen simultaneously: existing tools provide no facility for synchronizing the animations other than specifying the duration of each animation, and manually synchronizing their timings.

Approaches also exist based on traditional text-based mark-up languages, such as TeX and HTML (with modern versions of CSS.) These approaches tend to have a low level of abstraction, in the sense that the programmer must manually keep track of: (a) the current state of presentation, (b) which slide is currently being generated, and (c) the durations of all the parts of the animations such that they sync up.

1.3 Thesis

Listing 1.1: First example of a Thapl program

```
PLAY hello world
LOAD library\animable from "library.thapl"

DRAMATIS PERSONAE
hello (animable <"\node[([(_\keys)]) (\%(name)) (0,0) {((contents));">):

  HAS name = "hello"
  HAS contents = "hello, world!"
  HAS draw = "blue80"
  HAS line width = "2mm"

ACTION
  hello disappear
  hello appear
```

This Thesis presents Thapl, a theatrical programming language, that was designed from inception to tackle the problem of specifying the behavior of on-screen
objects in a readable, succinct manner, in order to make slide-show animations maintainable. THAPL is implemented as a text-based domain-specific programming language (DSL) in order to allow the use of standard software engineering tools, such as diff [MacKenzie et al., 2002] and source control [Torvalds and Hamano, 2005]. It is a preprocessor, generating code in a host language. This design allows relegating the duty of rendering drawings to a different program, making it possible to limit the scope of the language to handling behavior rather than handling the already difficult task of encoding illustrations.

The main idea that THAPL advocates is that by focusing on specifying behavior, using ideas taken from the declarative programming paradigm, allows presentation authors to create maintainable presentations. Using THAPL, the programmer describes the animation in a structured way, declaring relations between events. It is then the compiler’s responsibility to sort out the details, e.g., enumerating slides and the duration of each part of the animation.

In that sense, the difference between THAPL and other tools is much like the difference between “WYSIWYG” (what-you-see-is-what-you-get) tools and “WYSIWYM” (what-you-see-is-what-you-mean) tools in document editing; like WYSIWYM tools, THAPL aims to separate the presentation from the content, the content of animations being the behavior of the entities contained within them.

The primary characteristics of THAPL are a declarative approach to defining relations between different parts of an animated slide-show using a physics-inspired model based on springs, and a flexible, user-extensible syntax that resembles English, implemented using a data-structure named the Context. This document describes these features, along with a definition and discussion of THAPL’s syntax.

The structure of the language is explained in Chapter 4; examples of slide-shows constructed using THAPL, and demonstrations of solutions to real-world problems, are presented later in this document. The structure is inspired by theatrical plays, and more specifically the works of Shakespeare. The inspiration is evident in the way THAPL programs are divided into acts and scenes, and the separation of the “Dramatis Personæ” from the “action”.

An introductory example depicting the general shape of a THAPL program is presented in Listing 1.1. This is a variation on the classic “hello, world!” example made famous by C [Kernighan and Ritchie, 1988]: It creates an animation that displays the text “hello, world!” , which then disappears and reappears.

1.4 A Simple Example in THAPL

A more useful example of a THAPL program with non-trivial logic, is presented in Listing 1.2. This particular example draws several circles, and moves them around so that they intersect. The intersection is colored in a different color, and the different sections of the illustration remain consistently colored as they move around the screen.
A single frame of the output of the program shown in Listing 1.2 is depicted in Section 1.4. Each frame contains a bounding box and two circles, one red and one blue. When the circles intersect, the intersecting section is gray; when they exit the box, the fill is transparent. The drawing is accomplished using \LaTeX commands, not by \textsc{thapl}; in fact, \textsc{thapl} does not have commands for drawing on the screen, and always relies on \LaTeX (or another underlying language) in order to render graphics.

An important aspect of \textsc{thapl} is the structure that it enforces for its presentations. The root element of this structure is a \textsc{Play}, which in more complex animations can be subdivided using the \textsc{Act} keyword, which in turn can contain sub-animations that are defined with \textsc{Scene}.

Each animation may contain a variety of clauses. The first one that we encounter here, \textbf{Dramatis personae}, or \textbf{Actors} for short, is a list of actor entities (Dramatis Personæ) that can partake in actions in this act. The last clause in the example, marked
with the ACTION keyword, is an execution clause, and contains the steps necessary to create the actual animation. Each line in the action part is carried out in sequence, unless otherwise specified. For example, using the MEANWHILE operator allows actions to be performed in parallel. Performing actions in parallel and other time related topics are discussed in Chapter 6.

THAPL somewhat resembles a language in the imperative programming paradigm, but has no assignment statements that allow modifying state. Instead, THAPL uses CHANGE directives, which are the counterparts to assignment statements. CHANGE directives declare that a state change will occur over time, thereby creating an animation. Therefore, as a matter of terminology, the language does not contain variables nor procedures. Instead, THAPL has actors, characters, and actions: Actions can be performed on actors, and actors can inherit from characters, which are a sort of abstract actor. All these elements are then organised together in plays, acts, and scenes.

Syntactically, THAPL uses white-space and colons to denote blocks. Additionally, THAPL has several syntactic features intended to make the language more flowy and English-like, and to more closely imitate the look-and-feel of a dramatic play.

It is important to note that THAPL is a preprocessor language, i.e., it extends another language by wrapping around it. In Listing 1.2, this is obvious from the \LaTeX code that is passed as the argument animable’s constructor (also called rendering): it is this code that is rendered into the actual slide-show, which is then compiled by \LaTeX to produce the slide-show PDF that is to be presented.

**Outline**

The remainder of this document is organized as follows. Chapter 2 discusses existing state-of-the-art tools, and presents other solutions in the problem space. In Chapter 3 THAPL is introduced by way of example, specifically, by showing how an animation explaining the solution to the Towers of Hanoi puzzle can be created.

The following sections focus on the THAPL language. Chapter 4 formally defines the syntax of the language and its semantics. In Chapters 5 and 6 two major contri-
butions of this work are discussed. First is the Context data-structure, which is used to implement name resolution in the language, and second we develop constraint-based control of time inspired by the physical model of springs, which makes it possible to transform the declarative specification into renderable slides.

Having developed the THAPL compiler prototype, Chapter 7 validates its fitness for purpose and presents potential contributions by applying the language to real-world problems. Finally, the thesis is concluded in Chapter 8, which compares related research to THAPL, discusses possible future applications based on concepts introduced in this work, and offers concluding remarks.
Chapter 2

Background

In order to facilitate a productive discussion of THAPL, we must first review the current state of software and ideas in the domain of animation generation. THAPL has a dual purpose, being developed both to make a point on programming language styles and software styles in general, as well as to be an actual useful tool to generate animations. To fully appreciate the language, it is helpful to be aware of the niche it fills and its domain.

It is assumed that readers of this document are already at least somewhat familiar with popular software packages for authoring slide-show presentations, e.g., Microsoft PowerPoint. This chapter attempts to provide such an overview of that domain, and categorize the different applications in terms of domain-specific languages (DSLs [Mernik et al., 2005]). DSLs are programming languages which are designed to be used in a specific domain, and contain operations that are useful in the context of that domain; DSLs trade generality for expressiveness in a limited domain. A well-known example is the SQL language, which specializes in querying relational database systems. DSLs can be categorized into either an internal or external implementation style [Schauss et al., 2017]. An internal DSL is one which is embedded in a host-language, essentially being a library or framework in the host language, perhaps utilizing a fluent API [Fowler, 2010]. Meanwhile, an external DSL is not tied to a specific host language. In the context of making animations, external DSLs tend to fall into one of two categories: presentation software, or visualization languages.

Although THAPL’s useful purpose is producing slide-show animations, the solution to that problem represents a novel approach to programming. A robust review of related research of programming languages can be found in Section 8.1.

2.1 Presentation Software

In this category, we include software products that are explicitly designed to create presentations, though usually not specifically animations. There are many existing, well-known packages for authoring presentations, which are suited for the task they were
designed for. However, the difficulty presented by informational animations was not a primary design consideration, and therefore none of these solve that particular issue to this author’s satisfaction. The difficulties presented by authoring animations tend to stem from core design designs, be that they operate at too low a level of abstraction, are not well-suited for creating animations; or are aimed at authoring media of other types (e.g., films), and simply support slide-show creation as an after-thought.

**PowerPoint** [Microsoft Corporation, b] is an impressive application for creating presentations. It also features a DCOM interface, that allows for automation and the ability to create presentations and animations programmatically [Salam, 2015]. However, the DCOM interface is poorly documented and not user-friendly, though when used by an expert it is possible to create very impressive presentations. Additionally, PowerPoint can be controlled using the **Visual Basic for Applications** (VBA) programming language [Marcovitz, 2012]. VBA can be used to create animations in one of two methods: Either by programmatically creating a normal PowerPoint presentation, i.e., automating the process of authoring PowerPoint, or by using a PowerPoint slide as a canvas which can then be animated through a low-level interface. The latter method would essentially require writing rendering code for every animation.

Of particular note is PowerPoint’s **Animation Pane** [Microsoft Corporation, a], a graphical user interface for creating and editing animations. One may go as far as saying that the animation capabilities provided by the animation pane can be considered a primitive graphical programming language. Sadly, the animation pane is lacking; while it does provide a more structured way to control animations, it is error-prone, complex in its presentation, and makes modification of existing animations extremely difficult. There are also other, comparable software packages that utilize the same paradigm as PowerPoint, such as **Keynote** and **LibreOffice Impress**.

Listing 2.1: A simple animation in \LaTeX

\begin{frame}[fragile]
\frametitle{}
\begin{center}
\begin{tikzpicture}
\node [at={(1, 1)}] {};
\node [draw,at={(4, 4)}] 0;
\only<1> \node [shape=circle,fill=red,at={(1, 1)}] {};
\only<2> \node [shape=circle,fill=red,at={(2, 2)}] {};
\only<3> \node [shape=circle,fill=red,at={(3, 3)}] {};
\end{tikzpicture}
\end{center}
\end{frame}

\LaTeX{} [Lamport, 1994] is a well-known document preparation system built around the TEX mark-up language which is used for typesetting documents; in fact, this very document is authored and typeset by \LaTeX{} (specifically, by \XeLaTeX{}). In addition to being a DSL for type-setting documents, with the addition of a few packages it is
possible to create and animate presentations: Beamer [Tantau et al., 2018] is used to create slide-shows, providing an interface for creating documents where each page is a slide; TikZ [Tantau, 2019] enables the user to draw figures and diagrams, and the animate [Grahn, 2011] module provides a method to animate figures with a “play” button that starts the animation and plays the frames that is supported by some PDF readers. The main issue with creating animations directly in L\(\text{ATEX}\) is that it provides a very low level of abstraction, requiring the author to reason in term of slide numbers. While this allows full control of the resulting slide-show, allowing for unlimited design ability, nothing is easy. Consider Section 2.1, which is a three-slide animation moving a circle around. Note how the motion is controlled using \texttt{\only}, where the slide number is explicitly referenced. While in this simple case there are other approaches, explicit numbering is still the norm in more complex animations.

\(\text{T\text{\lowercase{E}}X}\) is a Turing-complete language; as a result, there are examples that compute the animations they produce, for instance the Towers of Hanoi implementation [Damman and Hofmann, 2010], included for convenience in Appendix B.1.1, computes the solution to the puzzle via a recursive algorithm then generates an animation from that computed solution. While visualizing an algorithm in action is an interesting endeavor and allows for flexibility, e.g., by changing the number of discs in the Towers of Hanoi animation and having the animation adjust seamlessly, the literature [Stasko et al., 1993] suggests that such animations do not in fact assist one in learning the algorithm. Instead, the animation should be designed to meet a specific instructional goal; this is more easily achieved by specifying the actions occurring in an animation, instead of displaying the work performed by the algorithm. Consider also the case where the instructor would like to animate a “wrong” step, i.e., to show something that the algorithm does not do, in order to explain the reasoning behind the algorithm. This idea is extendable to slide-shows in general, not just algorithm visualizations in particular. Therefore, Thapl does not attempt to be a Turing-complete language which can encode arbitrary algorithms and then visualize their behavior, instead focusing on specifying the behavior of on-screen entities, unlike these \(\text{T\text{\lowercase{E}}X}\) examples.

TikZ also provides a full-blown, stand-alone animation feature in the form of a library. These animations can then be compiled into SVG documents where they can be viewed as continuous animations, or discretized into slide-based animations using a snapshot feature. An example from the TikZ manual [Tantau, 2019] showing the “Moon” orbiting the “Earth,” which in turn orbits the “Sun,” is shown in Listing 2.2, and the resulting animation is visualized by taking two snapshots (a TikZ feature) at \(t = 0.7s\) and \(t = 2.2s\) in Fig. 2.1. This offers a more succinct and flexible way to program animations than the method described above; however, the composer must still define the animation in terms of start time and end time, and manually sequence sub-animations, rather than specify the behavior in terms of “before”, “after”, and “during”, as Thapl allows.

CSS [Birtles et al., 2018] is the styling language used in web design, and is supported
Listing 2.2: Animation using Ti\kZ's animation facility.

\begin{tikzpicture}
\node [color = {0s = "orange",
2s = "red",
4s = "orange",
repeats}]{Sun};
\begin{scope}[animate={orbit={2.5cm}{365}}]
\node {Earth};
\node [animate={orbit={1cm}{28}}] {Moon};
\end{scope}
\useasboundingbox (-3.8,-3.8) (3.8,3.8);
\end{tikzpicture}

Figure 2.1: Ti\kZ example visualized

by most modern browsers. It supports adding animations to web pages, and supports building graphically exciting animations. In Listing 2.3 we see an example of such an animation. The code shown adds an animation named \texttt{slidein} to the \texttt{p} element, with a duration of three seconds. After that the animation is controlled by using key-frames, that is by declaring the value of CSS properties at certain points of the animation, and allowing the browser to interpolate the intermediate values. Sequencing animations is done manually, by specifying an \texttt{animation-delay} property and delaying the start of the next part of the animation to a certain point in time. While powerful, this abstraction operates at a very low level; it is difficult to implement an entire animation using only CSS. For this reason, animation is more commonly programmed using \texttt{JAVASCRIPT} [Flanagan, 2011], or CSS preprocessors like \texttt{SASS} or \texttt{LESS}. \texttt{JAVASCRIPT} is a general-purpose language, and allows creating animations using imperative code; similarly, \texttt{SASS} provides a for-loop, which simplifies creating key-frames and staggering animations by scheduling delays. \texttt{LESS} helps by providing some additional syntax, but doesn’t assist in sequencing multiple animations.
2.2 Visualization Languages

A visualization language is a DSL that is intended to be written by hand or generated by a program, and describes an animation of some sort; in particular, an algorithm animation language describes the animation of an algorithm. Karavirta, et al. [Karavirta et al., 2010] curate a taxonomy of such languages, and compare them on several categories. This taxonomy was constructed in the context of educational research, i.e., in the eyes of the authors, the intended audience for algorithm animation languages are teachers of computer science and their students. Many of the algorithm animation languages examined in the taxonomy are the scripting language part of an algorithm animation system, which for example allows interleaving annotations within a host general-purpose programming language in order to generate animations during runtime. As such, these languages tend to be imperative and slightly low-level. The languages examined usually provide a pre-defined vocabulary of useful data-structures or shapes (e.g., stack, graph, array) and operations that can be performed on them. In general, low-fidelity graphics are considered satisfactory for the purposes of algorithm animation languages, simplifying the implementation.

AnimalScript [Rössling and Freisleben, 2001, Rößling et al., 2004] is one such language. There are two versions available; AnimalScript has a line-based parser, and is firmly an imperative programming paradigm language, as can be seen in the AnimalScript implementation of the Towers of Hanoi puzzle in Section 2.2 (compare the Thapl implementation in Chapter 3.) AnimalScript2 is a re-implementation of the language, using an AST-based parser that allows for structured programming elements such as loops. This is showcased by the visualization of the bubble-sort algorithm in Listing 2.5. It should be noted that duration of the animation in both AnimalScript and AnimalScript2 is specified explicitly, using code like within 10 ticks. There
Listing 2.4: Towers of Hanoi in ANIMALSCRIPT [Rössling and Freisleben, 2001]

```animal
\% Animal 1.4
{
polygon "ls" (35, 40) (35, 120) (45, 120) (45, 40) color black \-
  filled fillColor darkGray
polygon "cs" (110, 40) (110, 120) (120, 120) (120, 40) filled
polygon "rs" (185, 40) (185, 120) (195, 120) (195, 40) filled
polygon "d1" (30, 90) (30, 100) (50, 100) (50, 90) filled fillColor gold
polygon "d2" (25, 100) (25, 110) (55, 110) (55, 100) filled
polygon "d3" (20, 110) (20, 120) (60, 120) (60, 110) filled
}
{
polyline "p" (35, 90) (35, 20) (110, 20) (110, 110) hidden
  move "d1" via "p" within 10 ticks
}
{
polyline "p0" (35, 100) (35, 20) (185, 20) (185, 110) hidden
  move "d2" via "p0" within 10 ticks
}
# (... skipping 20 lines of code ...)
{
polyline "p5" (35, 110) (35, 20) (110, 20) (110, 90) hidden
  move "d1" via "p5" within 10 ticks
}
text "tText" "# Moves: 7" at (113, 140) centered font SansSerif size 16 bold
```

This sample is taken verbatim from the ANIMALSCRIPT paper.

Listing 2.5: Bubble sort in ANIMALSCRIPT2 [Rößling et al., 2004]

```animal
array "values" (10, 10) length 5 int "3" "2" "4" "1" "8"
int i = 4;
arrayMarker "i" on "values" atIndex i label "i"
int j = i;
for (; i >= 0; i--) {
  moveMarker "i" to position i within 5 ticks
  for (; j <= i; j++) {
    moveMarker "j" to position j within 5 ticks
    if (values[j-1] > a[j])
      arraySwap on "values" position i with j within 5 ticks
  }
}
```

This sample is taken verbatim from the ANIMALSCRIPT2 paper.

is support for statements occurring in the same animation step, using a block: i.e., by placing multiple statements between { and }. However, there is no facility for synchronizing various animation steps aside from manually specifying the animation duration and delay.

The other algorithm animation languages described in Karavitra et al. are generally comparable to ANIMALSCRIPT. JAWAA [Akingbade et al., 2003] has a set of primitive shapes, a small library of data-structures and an associated set of actions that can be performed upon these entities, in an imperative fashion. GRAPHXML [Herman and Marshall, 2001] is an XML-based language, which describes a graph, and then allows that graph to be animated by describing a series of edits, i.e., node replacements, in that graph. All the languages surveyed are based on sequential statements; certain ones also support other concepts from the imperative programming paradigm such as branching, loops, and subroutines, but most do not. The majority of the languages
surveyed support concurrency, however as in ANIMALSCRIPT, the timing support is limited to controlling the duration and delay. A more detailed comparison with THAPL can be found in Section 8.1.3.
Chapter 3

Tutorial by Example

Figure 3.1: The Towers of Hanoi, starting position

1. Only one disk may be moved in each turn.

2. Each move consists of taking the upper disc from one of the stacks and placing it on another rod, so that it rests on the top of that rod’s stack.

3. Each rod’s stack must be in ascending order of size at all times.

With three discs, it takes seven turns to solve the puzzle. There exist both iterative and recursive algorithms to solve the puzzle, but in this example we are not interested in encoding those algorithms in Thapl. Instead, we are interested in making an animation showing the moves the algorithm would have made as a means of visualizing the algorithm’s run.
3.1 Introduction

We begin with an empty program, as shown in Listing 3.1. When run in stand-alone mode (i.e., with \LaTeX integration turned off,) running this program produces no output.

This is a good starting point to explain how Thapl works. There are two interesting concepts at play in the empty program listed in Listing 3.1. First, we see that we have created a play, and named it hanoi. Second, we have defined a character called animable. The play is an organizing structure, similar to a class, or function, and a character is like an abstract class. The animable character is special because the rendering phase uses the contents of the render property to render actors; in other words, it is special because it has render. We have defined animable explicitly, but usually we load it from the standard library. This is shown in Listing 3.2, which shows the same empty program, but this time animable is imported from the standard library.

3.1.1 Defining characters

To begin with, let's define a character for a generic pole and a disc. This is shown in Listing 3.3.

Listing 3.2: Empty Thapl Program, with load statement

Listing 3.3: Definition of pole and disc

Of course, poles and discs are more than just empty definitions. Discs are defined by their width and position, and poles are defined by their position. In Listing 3.4 we add these properties. We also set rendering on the inherited animable using the constructor; rendering is a feature of the standard library animable which enables us to show and hide the actor. Since disc and pole both have a position, we unify
this field using a mix-in, i.e., using multiple inheritance, with another character named locatable.

### Listing 3.4: Adding has properties to pole and disc

```play
PLAY hanoi
LOAD library\animable, library\location from ".../library.thapl"
CHARACTERS
locatable:
  HAS position (location)
pole (locatable, animable "pole at (( position/x ))")
disc (locatable, animable "disc at (( position/x ), (( position/y )), width = (( width ))")):
  HAS width (real)
```

### 3.1.2 Defining actors

We have completed the definitions of the character types of the Hanoi program; the next step is to make the program actually output something. To do that, we need to define actors. In this case, six actors are needed in total: three poles, and three discs. These are defined in Listing 3.5.

### Listing 3.5: Defining actors for poles and discs

```play
PLAY hanoi
LOAD library\animable, library\location from ".../library.thapl"
CHARACTERS
locatable:
  HAS position (location <\but x = 0, y = 0>)
pole (locatable, animable "pole at (( position\x ))")
disc (locatable, animable "disc at (( position\x ), (( position\y )), width = (( width ))")):
  HAS width (real)
ACTORS
poles:
pole 1 (pole <\but position\x = 0.0>)
pole 2 (pole <\but position\x = 5.0>)
pole 3 (pole <\but position\x = 10.0>)
discs:
disc 1 (disc <\but position\y = 1.0, width = 40.0>)
disc 2 (disc <\but position\y = 6.0, width = 35.0>)
disc 3 (disc <\but position\y = 11.0, width = 30.0>)
```

Now that we have defined actors, our program has an output, shown in Fig. 3.2. However, note that this is just a single “frame” of our animation, the starting frame. For actual animations, that contain more than a single static frame, we require actions, which we will now describe.

### 3.2 Action

Listing 3.6 depicts a simple action statement, that simply moves disc 1 to the position of another pole. This is useful for illustrating what an action is (but it is not what we want, either from the language or from the animation.)
Listing 3.6: Adding a simple action to the animation

PLAY hanoi
LOAD library\animable, library\location from ".\library.thapl"

CHARACTERS
locatable:
  HAS position (location <BUT x = 0, y = 0>)
pole (locatable, animable <"pole at ((position\x ))">)
disc (locatable, animable <"disc at ((position\x )), ((position\y )), width = ((width ))">):
    HAS width (real)

ACTORS
poles:
  pole 1 (pole <BUT position\x = 0.0>)
pole 2 (pole <BUT position\x = 5.0>)
pole 3 (pole <BUT position\x = 10.0>)

discs:
  disc 1 (disc <BUT position\y = 1.0, width = 40.0>)
  disc 2 (disc <BUT position\y = 6.0, width = 35.0>)
  disc 3 (disc <BUT position\y = 11.0, width = 30.0>)

ACTION
CHANGE discs\disc 3\position\x to poles\pole 2\position\x

As mentioned, the major new concept in Listing 3.6 is the \texttt{CHANGE} statement. By default, when using \texttt{CHANGE} on a numeric field, the length of the mutation created is the size of the change requested. That is, when changing a field from 5 to 9, the change would occur over $9 - 5 = 4$ slides. This can be overridden by the user. For example, to change the field \texttt{foo} to 8 in just two slides, it is possible to use the following directive: \texttt{CHANGE foo to 8 in two slides}. Note the usage of English numeral literals in the example; \texttt{THAPL}'s support for these literals is described in Section 4.1.1.

In Fig. 3.3 we can see the output of the program. In the name of brevity, only the output for \texttt{disc 3} is included, since the rest of the output remains unchanged from Fig. 3.2. Since the change is to $x$, from 0 to 5, the change takes five slides, or six slides in total including the initial state.

```
disc at 0.000000, 11.000000, width = 30.000000
disc at 0.533333, 11.000000, width = 30.000000
disc at 1.833333, 11.000000, width = 30.000000
disc at 3.166667, 11.000000, width = 30.000000
disc at 4.466667, 11.000000, width = 30.000000
disc at 5.000000, 11.000000, width = 30.000000
```

Figure 3.3: Truncated output of the program shown in Listing 3.6
3.2.1 Defining verbs

The completed animation should show the disc moving up and off the pole that it is currently on, move to position above the target pole, and finally drop onto that pole. Additionally, this multi-step motion should be replicatable for all discs. For this purpose, can properties, or verbs, come in useful. These are like class methods—the class is the actor, and the can property acts upon it. The output of the program is not included here, as it is a little long and not dissimilar to Fig. 3.3.

Listing 3.7: Defining a CAN property

```
PLAY hanoi
LOAD library\animable, library\location from ".../library.thapl"

CHARACTERS

locatable:

HAS position (location < BUT x = 0, y = 0>)
pole (locatable, animable <"pole at (( position\x ))">)
disc (locatable, animable <"disc at (( position\x ), (( position\y )), width = (( width )))>):

HAS width (real)
CAN move:

[to $pole at height $height]
CHARGE position\y to (( $pole\position\y + 4.0 ))
CHARGE position\x to $pole\position\x
CHARGE position\y to $height

ACTORS

poles:

pole 1 (pole < BUT position\x = 0.0>)
pole 2 (pole < BUT position\x = 5.0>)
pole 3 (pole < BUT position\x = 10.0>)

discs:

disc 1 (disc < BUT position\y = 1.0, width = 40.0>)
disc 2 (disc < BUT position\y = 6.0, width = 35.0>)
disc 3 (disc < BUT position\y = 11.0, width = 30.0>)

ACTION
discs\disc 3 move to poles\pole 2 at height 1.0
```

Now that the main verb of the program is defined, we can encode the entire solution to the Towers of Hanoi puzzle, depicted in Listing 3.8. Note that the only difference between Listing 3.7 and Listing 3.8 is in the ACTION part; this is an example of how Thapl simplifies the process of specifying behavior.

3.2.2 Controlling time

Let us consider a situation where we would like like to stream-line the presentation by having two discs move at the same time: This is technically against the rules, but the animation is conveniently shortened by displaying two steps simultaneously. With Thapl, this is straight-forward: we use the MEANWHILE keyword, as seen in Listing 3.9.

3.3 Conclusion

So far we have described the behavior of a solution to the Towers of Hanoi puzzle. However, we have not actually drawn any graphics in \LaTeX/TikZ. The objective is
to put \LaTeX/TikZ code inside the rendering fields of the animables. Since this is a THAPL tutorial rather than a \LaTeX/TikZ one, this is not further explained; for more details, the completed animation is available in Appendix B.1.2.

The tutorial demonstrated the creation of a THAPL animated presentation from start to finish, showcasing the feature set of the language and its usefulness as a pre-processor for another language (\LaTeX). The basic usage of the language has been explained, namely characters, actors, defining actions, and specifying behavior. In the next section, we will formally define the language and its semantics.
Listing 3.9: Solving the Towers of Hanoi quickly using **MEANWHILE**

```play
LOAD library\animable, library\location from ".../library.thapl"

CHARACTERS
pole (animable : rendering = "pole at (( position\x ))");
   HAS position (location : x = 0, y = 0)
disc (animable : rendering = "disc at (( position\x )), (( position\y )), width = (( width ))");
   HAS position (location : x = 0, y = 0)
   HAS width (real)
   CAN move:
      [to $pole at height $height]
      CHANGE position\y to (( $pole\position\y + 4.0 ))
      CHANGE position\x to $pole\position\x
      CHANGE position\y to $height

ACTORS
poles:
pole 1 (pole : position\x = 0.0)
pole 2 (pole : position\x = 5.0)
pole 3 (pole : position\x = 10.0)
discs:
disc 1 (disc : position\y = 1.0, width = 40.0)
disc 2 (disc : position\y = 6.0, width = 35.0)
disc 3 (disc : position\y = 11.0, width = 30.0)

ACTION
discs\disc 3 move to poles\pole 2 at height 1.0
Meanwhile discs\disc 3 move to poles\pole 3 at height 6.0
Meanwhile discs\disc 2 move to poles\pole 3 at height 1.0
Meanwhile discs\disc 2 move to poles\pole 1 at height 1.0
Meanwhile discs\disc 2 move to poles\pole 2 at height 6.0
Meanwhile discs\disc 3 move to poles\pole 2 at height 11.0
```

```
```
Chapter 4

Syntax

This section explains Thapl’s unique and open syntax. There are many functional as well as stylistic choices encoded in the syntactical decisions made during the design and implementation of Thapl, and this is an attempt at explaining and rationalizing these decisions. The major goal the syntax attempts to accomplish is readability; it is critical that a presentation authored using Thapl be easy to understand. Another important goal is maintainability: making modifications to Thapl presentations should be straight-forward. Finally, the last goal the syntax attempts to accomplish is flexibility, i.e., Thapl should be able to encode as many presentations as possible.

The major inspiration for the syntax is Shakespearian stage plays, and Thapl attempts to emulate the basic structure and feeling of written plays in order to better serve its intended purpose—making slideshow-based animations. Thapl is a whitespace language, like Python [Lutz, 1996], and uses indentation to mark the start and end of blocks (a task for which \{ and \} or keywords like begin and end are used in other languages.) Thapl follows the same basic rules as Python in regards to when a block starts: an increase in indentation following a line ending in a colon marks a new block. Additionally, a newline usually marks a new declaration.

Throughout this chapter, the language grammar will be displayed as derivation rules. These take the following format:

\[
\text{Non Terminal} \Rightarrow \text{derived tokens and rules}
\]

Some conventions are followed to ease reading. Non-terminal names are displayed in italics, Like so. Tokens are displayed in a monospaced font face, as in this example, and keywords are highlighted in a bold face, e.g., THEN or CAN. Red colored text identifies meta information for the rule, e.g., repeated or optional tokens, or for grouping tokens with parenthesis, (as seen here). To denote that more than one option is possible, a vertical bar is used, like such: (this\|\or that\|\or maybe those\). A suffix is appended when a token is optional, can be repeated zero or more times, or can be repeated one or more times, as in these examples, respectively: maybe\text{\_opt}, (zero or more)\text{\_*},
Tokens \texttt{NEWLINE}, \texttt{INDENT}, and \texttt{DEDENT} tokens are not shown in order to improve readability, and instead the convention that after a colon (:) that is not inside parenthesis a new block may be started is used to implicitly provide them.

\texttt{THAPL} is an optional terministic language in regard to statement terminators (the period), and a separatist language in regards to commas.

\subsection{Lexical Matters}

\texttt{THAPL}'s tokenization rules are designed to support its syntactic similarity to English.

\begin{description}
\item[Comments] Comments are marked \texttt{\%\{like this\}\%}, beginning with \$\{ and ending with \%\}. Comments can be nested, making it possible to comment out sections of code without fear. For example, \texttt{\%\{this is a comment.\}\%}, and \texttt{\%\{this is a \%\{nested comment\}\% \}}.

\item[White-space and line terminators] White-space can be either a character defined by the Unicode standard [Unicode Consortium, 2019] to be white-space, i.e. members of the “Space\_Separator” (“Zs”) category, or one of the following characters: exclamation mark ’!’, question mark ’?’ , comma ’,’ , underscore ’_’, tab (U+0009), vertical tab (U+000B), form feed (U+000C), and zero-width no-break space (U+FEFF), as well as line-terminators.

Line terminators are one of the following: line feed (U+000A), carriage return (U+000D), line separator (U+2028), and paragraph separator (U+2029). A sequence of a carriage return followed by a line feed is treated as a single newline token. Note that not all characters treated as line breaks by Unicode [Heninger, 2019] are considered to be newlines in \texttt{THAPL}.

\item[Words] A token, or word, is a case-insensitive sequence of meaningful characters separated by white-space. To clarify the definition, words are:

\begin{itemize}
\item Case-insensitive, so \texttt{green}, \texttt{Green}, and \texttt{GrEeN} all evaluate to the same token.
\item Comprised of non-removed characters in this context. Characters which don’t have any other meaning are ignored, e.g.:
\begin{itemize}
\item the dash character - outside number literals and array specification,
\item the at-sign @,
\item the octothrope #,
\item any other character not specifically mentioned in Table 4.1.
\end{itemize}
\end{itemize}

Therefore, \texttt{gr\-e\-en}, \texttt{gr\#Een}– and \texttt{\-g\-re\@en} all evaluate as the same token.
\end{description}
### 4.1.1 Literals

**String literals** are delimited with double-quotes, like "string literal". They are used for text constants, for example,

*Hamlet* says "Hello, world!".

Additionally, multi-line strings are supported, as in **Python**. These are delimited using three double-quotes, for example:

*Juliet* says """"What's in a name? That which we call a rose By any other word would smell as sweet;"""".

**String expressions** are a feature of strings that enable the referencing of variables within strings, causing the entire expression to be substituted by its evaluated value. Such expressions are denoted with (( and ))). Note that the parentheses must not have white-space separating them, and that the parsing is eager. Therefore, ((( ))) is likely to cause issues. If it is required that a string expression be placed within parentheses, comments can be used as they are also recognized within strings, e.g., /*/*(( ... )))/*. The interior of a string expression is a valid **THAPL** expression (described later), including referencing properties, arithmetic, and so on. Unlike other languages, like **Kotlin** [Jemerov and Isakova, 2017] that support a similar feature, string expressions in **THAPL** are lazily evaluated. The evaluation and subsequent substitutions occur during rendering, not literal definition time, and according to both the scope the string is currently in and the scope it was defined in, in that order.
Number literals are extended to be English-like, where the following are recognized as number literals:

- Natural numbers, like 6, 42, and so on, including negative numbers like -313.
- Numbers in decimal notation, such as 5.5, -4.3, and so on.
- Natural numbers in English, like Four hundred and three, Seventeen, two dozen, and four score and seven. Numbers with a scale of up to a trillion are recognized (the maximum being 999,999,999,999,999), as well as fractions up to one ninetieth (1/90). Additionally, the number names dozen (12), score (20), and gross (144), are also recognized by the tokenizer, to allow numbers like “four score and seven”.
- Fractional numbers in English, like one third or three fourths.
- Roman numerals, for example MMXVII or III. In order to allow words like I and MV, only roman numerals that are longer than 2 digits (letters) are tokenized as numbers.

Natural numbers are tokenized in a greedy fashion, by using the pattern for number names in English: The names for 0–19 are single words, as well as the tenth place for numbers 20–100. Then there are scales: a hundred, thousand, and so on.

Other literals include the Boolean literals, true and false, as well as the value of the unit type, nil.

Expressions are delimited in Thapl using double-parenthesis (similarly to string expressions.) It is vital to include a space after the opening delimiter and before the closing delimiter, e.g., this is correct: (( 1+1 )), while this will fail to parse: ((1+1)). The following is supported within expressions:

1. Ternary operator: condition ? true case : false case
2. Boolean operators: this && that, this || that, !that.
4. Binary addition operators: +, -(the unary - is part of the token definition).
5. Binary multiplication operators: *, /, %.
6. Parenthesis: (, )

The priority of each operator is denoted by its placement in the list, where a position closer to the end of the list means it will be evaluated first.
4.1.2 Tokenization sequence

Tokenizing a THAPL program, involves the application of the following steps in order:

1. Tokenize into proto-tokens, which are tokens that split input source into separate tokens and recognize string literals that are delimited with ". Additionally, comments are removed, but no further processing is done—specifically, non-meaningful characters are still included and keywords are not treated uniquely.

2. Apply the number literal pattern matcher on the proto-tokens, which greedily applies the literal patterns on the proto-tokens to produce number tokens.

3. Create INDENT and DEDENT tokens from leading white-space, emitting an INDENT token where the amount of leading white-space is greater than in the preceding line and a DEDENT token where it is less.

4. Finally, remove all remaining non-meaningful (i.e., characters not listed in Table 4.1) proto-tokens to produce THAPL tokens.

For example, consider the following excerpt from a THAPL program:

heap arrows:
  HAS counter = 0
  arrow one (heap arrow):
    HAS bend left = forty five
  arrow two (heap arrow):
    HAS bend right = 45

This is first tokenized into proto-tokens, with tokenization occurring as usual; the token stream will be “heap”, “arrows”, “COLON”, “NEWLINE”, “SPACE”, and so on. At this point, the number literal pattern matcher will be applied to the token stream. This program will locate any English number literals in the token stream, in this case “one” and the sequence “forty” and “five”, and transform them into the numeric tokens “1” and “45” respectively. Following this step, the space proto-tokens are converted into INDENT and DEDENT tokens as needed; for example, the two SPACE tokens before HAS counter are converted into a INDENT token, but the SPACE tokens in the following lines are ignored; the SPACE tokens in the line beginning with arrow two are converted into a DEDENT token, since they represent a reduction in indentation compared to the previous line. Following the end of this excerpt, assuming that this is the end of the program, two DEDENT tokens will be inserted. At this point, tokenization is concluded, and any remaining tokens are considered to be THAPL tokens and not proto-tokens.

Note that string expressions were not mentioned in this process; they are treated as regular strings, and are only evaluated during rendering.
4.2 Program Structure

We will now explore the syntax of a THAPL listing and formally define its construction. A basic sketch of the structure of a THAPL program is displayed in Listing 4.1. A THAPL program is composed of any number of \textit{plays},

\[
\text{Program} \Rightarrow \text{Play}^* 
\]

where a play is introduced by keyword \texttt{PLAY}, followed by the play’s name (which is a \textit{noun}), followed by a \textit{script}.

\[
\text{Play} \Rightarrow \texttt{PLAY \ Name \ Script} 
\]

This script is a fundamental unit of execution in THAPL, comparable to a procedure in imperative programming languages. It consists of two major parts: definitions and directives. Both parts are optional.

\[
\text{Script} \Rightarrow \text{Definitions}_{\text{opt}} \ \text{Directives}_{\text{opt}} 
\]

In order to organize the program in a hierarchical manner, plays can contain nested sub-parts called \textit{acts}, which in turn can contain nested \textit{scenes}. More generally, the program itself can be thought of as an anonymous global divisional unit, which may or may not contain plays. This extends the above definitions into their complete definition:

\[
\begin{align*}
\text{Program} & \Rightarrow \text{Play}^* \ \text{Script} & (4.1) \\
\text{Play} & \Rightarrow \texttt{PLAY \ Name \ Script} & (4.2) \\
\text{Act} & \Rightarrow \texttt{ACT \ Scene}^* \ \text{Name \ Script} & (4.3) \\
\text{Scene} & \Rightarrow \texttt{SCENE \ Name \ Script} & (4.4) \\
\text{Script} & \Rightarrow \text{Definitions}_{\text{opt}} \ \text{Directives}_{\text{opt}} & (4.5)
\end{align*}
\]
Plays, acts, and scenes are means for hierarchical, modular division of the program, similarly to classes, modules and routines in other languages. Each such unit begins with its own designated keyword, followed by a (possibly multi-word) name. Next comes the definitions section, which defines characters and actors. Entities defined in a play are recognized in all its acts, and entities defined in an act are defined in all its scenes. The directives chapter is optional, and if it is missing, the play is executed by running its acts in the order that they are written. Similarly, if an act has no directives chapter, the scenes are performed in the order they were written. However, if the directives chapter exists, then execution of the inner divisions is not performed implicitly. Instead, any such act or scene must be explicitly called. This design explicitly limits the level of nesting to three. This is justified in this domain, as the purpose of slide-shows is to be displayed to people in presentations, and the length of the human attention span acts as a natural barrier to slide-show length.

Each divisional unit (global, play, act, or scene) may contain chapters (recall Eqs. (4.1) to (4.4)). The definitions chapter is used to introduce two types of entities—characters and actors. Both are defined using the same syntax, but characters are abstract archetypes, similar to abstract classes in other languages, while actors are concrete stage players which will be rendered in the produced animation. THAPL’s divisional units are also similar to Pascal [Wirth, 1971]’s procedures in the way they are specified: like procedures, each divisional unit defines a name, variables (actors), types (characters), and a body (action).

The definitions chapter is divided into the characters and actors chapters, each of which begins with its own unique keyword(s).

\[
\text{Definitions} \Rightarrow \text{Characters}_{opt} \text{ Actors}_{opt} \quad (4.6)
\]
\[
\text{Actors} \Rightarrow (\text{DRAMATIS PERSONAE} | \text{ACTORS}) \text{ Actor Specification}^* \quad (4.7)
\]
\[
\text{Characters} \Rightarrow \text{CHARACTERS} \text{ Actor Specification}^* \quad (4.8)
\]

Similarly, the directives chapter is identified by the keyword ACTION, which is then followed by a (typically compound) directive.

\[
\text{Directives} \Rightarrow \text{ACTION} \text{ Directive} \quad (4.9)
\]

Note that the beginning of divisional units and the chapters they contain is marked with a keyword. White-space and indentation are used to mark the beginning and end of these blocks, and the markers for beginning and ending these blocks are elided in the syntax derivation rules shown here. An interesting by-product of the rules described here is that a divisional unit ends either at the end of input, or at a keyword marking a divisional unit at the same or a higher level. For example, a scene would end at the
beginning of a subsequent scene, and also at the beginning of an act or a play. Chapters end at the beginning of the next chapter, or at a keyword starting a new divisional unit.

The program is parsed into a Context, with each divisional unit and actor specification being described as a nested context. The run-time behavior and semantics of this object, as well as the exact manner in which Thapl concepts are expressed as contexts, are defined and discussed in Chapter 5.

4.3 Declarative Definitions

Thapl provides a declarative syntax to define new entities. This includes actors, which are rendered on screen and exist at run-time as independent units, and characters, which are “abstract actors” that will not be rendered into the final output on their own, and who’s main purpose is to be extended by an actor. Both actors and characters are specified using an entity definition; the knowledge of whether an actor or character is being defined is gleaned from the location of the definition (in either the actors chapter or a characters chapter, respectively). The entity definition consists of the entity’s name (a noun), a primary constructor clause defining the parameters with which an entity can be initialized, optional inheritance clause, and an optional ordered list of properties. The entity can also be defined as an array, with a distinct list of indices.

\[
\text{Entity Definition} \Rightarrow \text{Name} \ (\text{Array Spec})^\text{opt} < \text{Constructor} > \text{Inheritance List}^\text{opt} \ ( : \text{Property}^+)^\text{opt}
\]

\[
\text{Inheritance List} \Rightarrow (\text{Inheritance Clause} (, \text{Inheritance Clause})^*)^\text{opt}
\]

\[
\text{Constructor} \Rightarrow \text{Constructor Parameter} (, \text{Constructor Parameter})^*
\]

\[
\text{Inheritance Parameter} \Rightarrow \text{Parameter} (\text{Type})
\]

\[
\text{Array Spec} \Rightarrow [\text{Array Spec Item} (, \text{Array Spec Item})^*]
\]

\[
\text{Array Spec Item} \Rightarrow \text{Number} (.. \text{Number})^\text{opt}
\]

Actors can be defined to extend one another, using prototype inheritance in the style introduced by SELF [Ungar and Smith, 1987] and popularized by JavaScript. When defining an entity that extends other entities, properties from the base entity (or entities) can be initialized on the same line, as is done in Kotlin. Additionally, as a last resort option (since it breaks encapsulation somewhat,) we can use “but” overrides to set specific values.

\[
\text{Inheritance Clause} \Rightarrow \text{Supertype Name} < \text{Constructor Call} ((, \text{BUT} , \text{Override})^\text{opt})^>
\]

\[
\text{Constructor Call} \Rightarrow \text{Constructor Call Item} (, \text{Constructor Call Item})^*
\]

\[
\text{Override} \Rightarrow \text{Inheritance Key Value} (, \text{Inheritance Key Value})^*
\]

\[
\text{Constructor Call Item} \Rightarrow \text{Value} | \text{Inheritance Key Value}
\]

\[
\text{Inheritance Key Value} \Rightarrow \text{Key} = \text{Value}
\]

An important part of each entity is the list of properties it contains. An empty entity is not particularly interesting (though an entity with an empty property list is not
superfluous if it has an inheritance clause.) There are four types of possible properties:

\[
\text{Property} \Rightarrow \text{Has Property} \mid \text{Can Property} \mid \text{Nested Entity}
\] (4.16)

The \textit{has property} defines a new “field” the entity \textit{has}, explaining the title. It consists of a name and a type, where the type is specified by an inheritance clause, giving the possibility of initializing the type. Alternatively, the field can be initialized with a value, either a literal, a reference to a previously defined \textit{has} property, or a reference to a name defined in the primary constructor. When initializing references from a previously defined property, the reference can either be initialized to point to the value or to the property; the first is accomplished using the = initialization syntax, and the latter using =>. If the property is initialized, the type is optional as it can be inferred.

If the property is marked using the \texttt{meta} keyword, it will not be rendered unless explicitly referenced. That is, since Thapl is designed to be used alongside \LaTeX, \textit{has} properties are commonly used to represent PGF keys for use with TikZ, and string expressions can be used to render all the properties an actor contains. Properties marked as \texttt{meta} will not be rendered in the string expression in that case, and are meant to be used for internal book-keeping within the actor that should not be rendered to TikZ.

\[
\text{Has Property} \Rightarrow \textit{HAS} \texttt{meta}_{\texttt{opt}} \text{Property Name} \left( \text{Inheritance Clause} \right) \\
\mid \textit{HAS} \texttt{meta}_{\texttt{opt}} \text{Property Name} \left( \left( \text{Inheritance Clause} \right)_{\texttt{opt}} \left( = \mid => \right) \text{Value} \right)
\] (4.17)

In order to define actions that an actor \texttt{can} perform, including possibly transforming itself, we use a \textit{can} property. This defines a \texttt{verb} which can be used on this entity—which is a noun. This concept is analogous to methods in other languages. The basis of a \texttt{can} property is the name, which is followed by a variables declaration, modifiers if any exist, and then a directive defining the action to take when this verb is executed. \texttt{Modifiers} consist of a “glob”, a sentence like description containing \texttt{variables} that are marked by being prefixed by \$. A modifier can also have an optional directive, which is executed only if this modifier is encountered in the sentence. The usage of verbs and modifiers, including the definition of the “glob”, is discussed in Section 4.4.

\[
\text{Can Property} \Rightarrow \texttt{CAN} \text{Property Name} : \text{Can Variables}_{\texttt{opt}} \text{Modifier}^* \text{Directive} \\
\text{Can Variables} \Rightarrow \{ \text{Can Variable} , \text{Can Variable} \}^* \\
\text{Can Variable} \Rightarrow \text{Variable Name} \left( \text{Type Name} \right) \\
\text{Modifier} \Rightarrow \left[ \text{Modifier Glob} \right] \left( : \text{Directive} \right)_{\texttt{opt}}
\] (4.18, 4.19, 4.20)

We can also create a nested entity. Nested entities can be addressed en masse using the name of the nesting entity. The syntax for a nested entity is similar to the previously
defined entity definition, differing only in location:

\[
\text{Nested Entity} \Rightarrow \text{Entity Definition} \quad (4.21)
\]

Bringing it all together, in Listing 4.2 we see an example of a simple entity. Recall that we previously saw a verb definition in Listing 3.7.

Listing 4.2: An example of a simple entity definition.

```plaintext
record (node<> : name part\visible = true):
  has font = "\scriptsize\ttfamily"
  has fill = "blue!30"
  has draw (flag) = true
  can mark:
    \%( placeholder for verb definition (directives) \%)
  can unmark:
    \%( placeholder for verb definition (directives) \%)
```

4.3.1 Type System and Primitive Types

In order to facilitate a germane discussion on the syntax of THAPL, it is useful to offer a short description of the type-system. THAPL has an object-oriented type hierarchy, with the root, or top, type being Any. There are two types whose superclass is Any: Compound, which is the superclass of all user-defined entities (actors and characters), and Atomic, which is the superclass to the seven primitive types:

1. **Real**, representing a machine-level double-precision [Zuras et al., 2008] floating-point value,

2. **Integer**, an integer numeric value with an unlimited range (subject to available virtual memory),

3. **String**, 

4. **Unit**, a type with a single value named nil,

5. **Boolean**, a type with two values (true & false),

6. **Flag**, like Boolean but represents the existence of the variable instead of a true/false value; i.e., during rendering, a variable of type Flag would either render the variable’s name or nothing. Contrast Boolean, which will be displayed as either true or false.

7. **Reference**, a pointer type, containing a reference to an object of Any type or to a lazily-evaluated expression (such as the name of a has property).

Most of these types are straight-forward value types which require no further discussion. However, the Reference type, which allows referencing another value, should
be expanded upon. It can either point, by name, to another entity, or contain a lazily evaluated expression which may itself contain by name references to other entities. When passing this reference value as a parameter, we would like the references made to remain valid. This is realized by having the Reference close over its containing scope. We would also like to support dynamic dispatch so that inheritance works as expected, therefore the implementation uses a trampoline-like construction: each Reference contains a pointer to its enclosing scope (i.e., the Compound it is embedded in), and each Compound contains a downward pointer to its inheriting Compound. When resolving this pointer, it is snapped such that it points at the bottom-most object.

4.4 Directives: Sentences, Subjects, Verbs, and Modifiers

Directives are the THAPL construct most similar to statements in other imperative languages; however, they are not imperative statements as they do not describe the control flow of the language. They are the means by which one describes the action occurring in the animation, such as objects moving, appearing, disappearing, changing colors, and any other transformation. These directives feature properties from both the imperative and declarative paradigms; like imperative statements, a directive (usually) changes the state of the program, but like declarative statements the control-flow of the program is not directly modified by a directive. Unlike many other programming languages, THAPL has no loops, nor conditional control (outside expressions.)

Directives come in two varieties: atomic and compound,

\[
\text{Directive} \Rightarrow \text{Atomic Directive} | \text{Compound Directive},
\]

(4.22)

where (as is custom) a compound directive is constituted of other, compound or atomic, directives, while the semantics of atomic directives accords with the etymology of their name, i.e., they are indivisible into other directives.

4.4.1 Compound Directives

The default composition of directives, i.e., the way that directives are composed when no specific keyword is used and the directives are each put on their own line, is sequential:

\[
\text{Compound Directive} \Rightarrow \text{Sequential Directive}
\]

\[
\text{Sequential Directive} \Rightarrow (\text{Directive} \text{THEN}_{\text{opt}})^+ \]

(4.23)

e.g., Royalty quickly exits stage left \textbf{THEN} Hamlet moves to cue., or this equivalent multi-line example:

Royalty quickly exits stage left;
Hamlet moves to cue.
in which the \texttt{THEN} keyword is given using the `;` keyword instead. The period keyword `.` performs the same action. In fact, merely beginning the directive on a new line achieves the same end effect; the various keywords are used here to improve readability, or as a stylistic choice.

Composition of directives can also be \textit{in parallel}, i.e.,

\begin{align}
\text{Compound Directive} & \Rightarrow \text{Sequential Directive} | \text{Parallel Directive} \\
\text{Parallel Directive} & \Rightarrow (\text{Directive MEANWHILE})^+ 
\end{align}

However (unlike sequential composition) the \texttt{MEANWHILE} separator keyword is mandatory. The exact semantics of THAPL directive composition are discussed in Chapter 6.

4.4.2 Atomic directives

An atomic directive is comparable to a simple sentence in a natural languages—a list of words that includes in it a subject, a verb, and a number of adjectives, adverbs, which are collectively dubbed \textit{modifiers} in THAPL. However, it is important to note that THAPL is not a natural language, and is in fact a \textit{constructed language}, or a formal grammar designed to read like a natural language, but which is not written like one.

\begin{align}
\text{Atomic Directive} & \Rightarrow \text{Sentence} \\
\text{Sentence} & \Rightarrow (\text{Subject ,})^{+} \text{ Modifier}^{*} \text{Verb Modifier}^{*} 
\end{align}

As one can see, the structure of the sentence is flexible, but not unconstrained. Each sentence directive begins with one or more subjects, and contains exactly one verb. Between the subjects and the verb there can be any number of modifiers, as well as after the verb. There is no difference in placing a modifier before or after the verb—this feature exists to increase readability, e.g., by allowing writing \texttt{The Man jumps barely} as well as \texttt{The man barely jumps}. Both of these mean the same thing, but the author can select the version that better conveys her meaning to the reader. In common use we expect that certain modifiers will be usually placed before the verb, while others will be placed after it. The following are some examples of stylistic choices that apply to modifier location:

1. Both cases can be valid English: \texttt{candle slowly flickers} vs. \texttt{candle flickers slowly}.

2. One modifier placement might always make more sense: \texttt{computer changes color to green} vs. \texttt{computer to green changes color}.

3. The choice of verb might affect the placement of the modifier: \texttt{square rotate left} vs. \texttt{square left rotation}.
The subject, the verb, and the modifiers can each consist of multiple tokens. For subjects and verbs, these tokens are their names, but for modifiers, the tokens include parameters and are identified using pattern matching. We require that the subject appear first in order to minimize ambiguity. Although it would be possible to have the subject in the middle, as sometimes occurs in English, and then search the entire token sequence for tokens that name a known subject, this would cause ambiguity when naming other entities in a sentence. For example, in the sentence directive \texttt{Ball four changes color to Ball three\color}, both \texttt{Ball four} and \texttt{Ball three\color} are valid entity names. If the subject were allowed to appear anywhere in the sentence, it would create ambiguous sentences that are impossible to decode. While it is possible to set rules like “the first entity name in a sentence directive is the subject”, this is confusing to the programmer and it is simpler to have the subject appear in the first part of the sentence.

Parsing is carried out as follows: first \texttt{THAPL} searches the current scope for the subject; this is typically a name in the characters or actors section, but may also be a list of such names. The search is based on the knowledge that the first tokens in the sentence are subjects. The words that comprise the name are then marked as handled, and the search continues on to find the verb, which is a word or a list of words which identifies an action that the subject \texttt{can} carry out. These identifier words are also marked as handled, and once more the search continues for verb modifiers. Each of the verbs modifiers are pattern matched against the list of remaining words, erasing matched words until none remain. If there are any remaining words that cannot be matched by a known modifier, a syntax error is produced.

All the searches performed in the parsing process are \textit{greedy}, that is, they attempt to find a match that is as long and specific as possible. For modifiers, a match that matches more words that explicitly appear in the pattern is preferred over a longer match. In the case that more than one match exists that of the same length and specificity, the one that was defined first is selected.

As mentioned, the location of modifiers before or after a verb is not meaningful, but the order of a modifier within the modifier list is. We can use this feature to create sequences using sentences.

\section*{Other Atomic Directives}

In addition to sentences, there are directives with a stricter syntax, which perform basic functions in \texttt{THAPL}.

\textbf{Change directive} This is used to transform a \texttt{THAPL} entity. It accepts a \texttt{THAPL} subject and changes it to another value. This directive creates a spring based on the transformation requested, so the transformation occurs over time. This directive can only operate on properties that have a primitive type; the type of mutation that
occurs is dependant on the type. So, for example, a property of type \texttt{Integer} changes gradually over time, while a property of type \texttt{String} will change at the midpoint of the mutation. The natural length of the mutation and the spring constant ($k$) of the attached spring (i.e., how resistant to change this \texttt{CHANGE} is) can be given using the optional parts of the syntax. If these are not given, then the natural length of the animation is set to the absolute value of the difference between the old value and the new value, and the spring constant is set to 1. For more information on the spring model, see Section 5.3.1 and Chapter 6.

\begin{align}
\text{Change Directive} \Rightarrow \text{CHANGE Subject to Literal or Reference} \\
& (\text{in Literal or Reference slides})_\text{opt} \\
& (\text{with } k \text{ of Literal or Reference})_\text{opt}
\end{align} 

Set directive Similar to the change directive, but not gradual mutation occurs, and in fact a spring is not created for this directive. This is useful for setting values that are not to be rendered, such as setting properties in verb modifiers.

\begin{align}
\text{Set Directive} \Rightarrow \text{SET Subject to Literal or Reference}
\end{align} 

Relax The relax directive is a \textit{no-op}, with a spring that is initially instantaneous, but is malleable so that when it occurs in parallel with another directive, it stretches to fill the required space. That is, by default it is created with a spring with an infinitesimal spring constant.

\begin{align}
\text{Relax Directive} \Rightarrow \text{RELAX}
\end{align} 

Call directive This is used to execute a nested divisional unit.

\begin{align}
\text{Relax Directive} \Rightarrow \text{CALL Name}
\end{align} 

4.5 Practicalities and Ambiguities

4.5.1 Parsing

THAPL can be described as a context-free grammar using a notation such as EBNF, but the resulting syntax would not only be large and cumbersome but also inherently incomplete, as the way \texttt{ACTION} sections are parsed depends on the entities defined in the relevant \texttt{DRAMATIS PERSONAE} (Dramatis Personæ) sections. In particular, consider the following non-sense sentence:
Without knowing how the entities were defined, it is impossible to parse this sentence. We know that \textit{Meep} is part of the subject; but the subject can be just \textit{Meep}, or \textit{Meep moop}, or even every word in the sentence except for \textit{doo}, which would be the verb. After that, it is impossible to know which token is the verb, and how the rest of the token stream is split to modifiers without first consulting the definitions. Consequently, while we did define an EBNF directive describing the structure sentence directives in Eq. (4.26), that rule does not actually aid in parsing.

Therefore, THAPL’s parser is a implemented as an LALR(1) parser, which is then augmented by a parser module that recognizes sentences according to a user-defined parser state. Parsing is performed by performing these steps:

1. Parse THAPL tokens using an LALR(1) parser that recognizes entities that are not sentence directives, including the basic structure as defined in Section 4.2, entity specifications as seen in Section 4.3, and the non-sentence directives from Section 4.4.2. No binding is done at this stage.

The output from this step is an AST representation of the program as a single \textit{compound directive}, with sentence directives represented as nodes containing unparsed token-streams, as well as a top-level context that holds both entities and runnable subsections.

2. Transform the AST into bytecode-like instructions:

   (a) Perform binding according to previous declarations (as stored in the Context, which is defined in the next chapter). For sentence directives in the action parts, the token-stream is also parsed at this time using the binding information from the context.

   (b) Simplify single directives that apply to multiple actors into multiple instructions that each apply to a single actor.

The output is a tree of bytecode instructions, similar to the AST, that represents a parsed program ready for execution.

At the instruction level, most binding has already been performed, though not necessarily all of it. For example, string expressions may contain references to entities that cannot be bound until run-time, since as mentioned earlier, string expressions are only evaluated at rendering time.

4.5.2 Ambiguities

As demonstrated in Section 4.4, there may be multiple ways in which a sentence can be parsed, and THAPL chooses the longest and most specific match to provide well-defined behavior. However, even though the behavior is well-defined given a set of entities, it
can result in a surprising result if read naïvely. For example, consider the following sentence:

Hamlet stabs Laertes with a poisoned sword.

A standard English reading will tell us that the subject is Hamlet, the verb is stabs, and so everything else must be modifiers. But the THAPL parse doesn’t use any English rules, instead using the entities defined by the author, who might have chosen to define an actor named Hamlet stabs, which has a verb named poisoned sword, completely changing the way the sentence is parsed.

This freedom, and consequent danger, is very much the intention. I imagine the language to be used with a carefully designed “theme library”, which defines useful characters and verbs for a certain theme, and for each animation to be of medium size, and straight-forward, much like a screen play. In such a setting, these ambiguities can be avoided by the theme designer, who can use THAPL’s behavior of choosing the longest and most specific match to make sure that the correct action is selected. Tooling may also assist in warning when ambiguities are likely to arise.

As always, additional flexibility increases the responsibility of the programmer to use the available features correctly. It is the author’s burden to choose good names for subjects, verbs, and modifiers, in order to create a readable and understandable animation—hopefully by extending an existing theme that already has reasonable conventions.
Chapter 5

Semantics: Language and Animation

This chapter discusses the semantics of the Thapl language, and introduces the central implementation technique used, the Context data-structure. Most of the information described here is not novel, and in fact is common in programming language design and implementation. However, the semantics of the language serve as an important backdrop to the novel language features. Novel ideas shown here include the Context data-structure, described in Section 5.2. The Context data-structure is useful beyond the scope of Thapl itself: this is discussed in Section 5.2.2. The mechanism by which Thapl translates the declarative descriptions of animations to actual presentations is described in Section 5.3.

5.1 Object Model

As mentioned earlier, in particular in Section 4.3.1, Thapl’s object model is an object-oriented, prototype-based scheme similar to that found in the SELF and JavaScript languages. In the Thapl object model, the superclass for all user-defined entities is the Compound. User-defined entities in Thapl include, as described in Chapter 4, actors, characters, properties, and the structural units themselves (plays, acts, and scenes, as well as the global unit). Since Thapl is a prototype inheritance language, there is no differentiation between classes and instances, and Compound is just a regular object. Instead of classes inheriting from other classes and values being instances of classes, in prototype inheritance, instances inherit directly from one another. In addition to Compound, there is also Atomic, which is the superclass of all primitive values (e.g., Real, Integer, ...). See also Section 4.3.1.

As an example, Listing 5.1 defines an object named record, which inherits from node, and the inheritance chain ultimately terminates with Compound. record contains properties, such as font and fill, both of which are an instance of String, which is a subclass of Atomic, and represent a primitive value. There are also can properties,
which are represented as instances of \texttt{Compound}.

Scopes in \texttt{Thapl} are also represented as objects, and addressed similarly to other entities. The listing defines a \texttt{PLAY} named \texttt{example}, which is an instance of \texttt{Compound}. It is contained within the global scope, which is also an instance of \texttt{Compound}.

5.1.1 Object Creation

Entities in \texttt{Thapl} are defined ahead-of-time, and there are no directives that explicitly create an entity. However, this does not mean that \texttt{Thapl} does not support dynamic object creation. Every time that a \texttt{can} property is called, or a divisional unit is executed using the \texttt{call} directive, any entities that are declared within the executed element are re-instantiated.

Entities are instantiated by creating a new, empty \texttt{Compound}, and populating it with the items in the entity specification. Inherited entities are copied (recursively) and then added to the entity.

5.1.2 Inheritance

\texttt{Thapl}, in a manner similar to \texttt{C++} using non-virtual inheritance, supports multiple inheritance. If entity \texttt{X} inherits from entities \texttt{A}, \texttt{B}, and \texttt{C} in that order, then searching for some method \texttt{M} in \texttt{X} would first search \texttt{X}, then \texttt{A}, then \texttt{B}, then \texttt{C}. \texttt{Thapl} does suffer from ambiguities in the case of a diamond inheritance graph, i.e., if \texttt{A} and \texttt{B} both inherit from \texttt{C}. The language does not currently provide any mitigations against the issue; this is a flaw which can be remedied in a future version.

5.2 Name Resolution in \texttt{Thapl}

The semantics of name resolution in the \texttt{Thapl} language are the same for \textit{scopes} and \textit{objects}; that is, the rules for looking up a name of an actor in the current scope are the same as the dynamic dispatch rules for looking up a \texttt{has property} in an actor. The nameable user-defined entities in \texttt{Thapl} were defined in Chapter 4; these include:

Listing 5.1: Thapl program demonstrating the object model.

```
PLAY example
ACTORS
   record (node<> : name part\visible = true):
      HAS font = "\scriptsize\ttfamily"
      HAS fill = "blue!30"
      HAS draw (flag) = true
      CAN mark:
         CHANGE fill to "red!30"
      CAN unmark:
         CHANGE fill to "blue!30"
   ACTION
      record mark
```
actors, characters, can and has properties, and the structural units (plays, acts, scenes, and the global unit).

The name resolution rules in THAPL follow lexical scoping rules, with dynamic dispatch when looking up actions on actors and characters. Names are resolved using the scope of the sentence, possibly searching any parent scopes, including inheritance. The rules are as follows:

1. Name resolutions in the **ACTION** part of divisional structures search the current scope, and any parent divisional structures, in order,

2. Name resolutions in entity verbs are resolved by consulting the following resources, in order:
   
   (a) The current entity’s names list;
   
   (b) The inheritance tree of the entity;
   
   (c) The scope of the containing scope, which may be the divisional structure or a nesting entity;

3. When searching the inheritance tree of the entity, the search will not try the containing scope of a parent entity; thus when seeking a specific field within an entity using the scope resolution operator, only names in the entity itself and its inheritance tree would be considered, and not names in the containing scope.

5.2.1 Implementation Using Context Data-structure

A **Context** is a mapping, similar, e.g., to a **PYTHON** dictionary, but with the following modification: while **PYTHON** dictionaries are composed solely of key-value pairs, a Context’s elements can also include nested Contexts. An illustration of this concept is shown in Listing 5.2. The idea is inspired by **LISP** [Graham, 1995] a-lists [McCarthy and Levin, 1965], and is used to implement all name resolution tasks in the THAPL language, including: static and dynamic scoping, inheritance, composition, and grouping. This is similar to how, in **PYTHON**, the dictionary is the main building block of name resolution, but the semantics of name resolution are encoded in the data structure itself instead of the supporting code, producing a flexible implementation.

Unlike a regular dictionary, which is normally implemented as a hash-map, a Context is organized as an ordered list. This allows for linear search, with prioritization between duplicate elements. The Context model is an extension of scope-graphs [Konat et al., 2013] [Neron et al., 2015], with the new feature of the graph’s edges being ordered. This modification enables the implementation of multiple inheritance semantics, which the original scope-graph model does not support.
Listing 5.2: Illustration of the Context concept using Python syntax

```python
{
  "a": 2,
  "b": 3,
  {
    "f": 7,
    {
      "x": 17,
      "t": 1,
    },
  },
  {
    "f": 3,
  },
}
```

**Context Operations**

The principal operation that can be performed on the Context is *lookup*. For example, in Listing 5.2, looking up “a” in the structure would return 2, and looking up “x” returns 17. In the case of duplicate values, searching for “f” returns 7 and not 3: This is because 7 appears first in the structure, and as mentioned, Contexts are ordered.

Other Context operations concern constructing new Contexts. These include inserting a value at a location in the Context, cloning a Context, and so on. There are helper operations that are useful in for implementing Thapl semantics; e.g., for handling the semantics of the scope resolution operator that looks up nested objects. Additionally, to handle grouping semantics, the Thapl implementation supports returning multiple results when looking up a name. That is, when dealing with groups, searching for “f” should return both 7 and 3, in that order.

**Groups**

Nested entities (defined in Section 4.3) can be addressed en masse using the name of the entity that contains them. For instance:

```
PLAY grouping
  ACTORS
    group:
      first (node)
      second (node)
      third (node)
  ACTION
    group hide
```

In this example, *first*, *second*, and *third* will be displayed for a single frame, and then all of them will be hidden, since the *hide* directive would be applied to all three. The representation of this Thapl presentation using Contexts is illustrated in Listing 5.3. Since *group* contains nested entities, it is considered an *aggregate* Context, and any directives applied to it are also applied to its nested entities.
Listing 5.3: Context construction for “play grouping” using PYTHON syntax

```
{
    "play grouping": {
        "group": {
            "first": { ... },
            "second": { ... },
            "third": { ... },
        },
        ...,
    },
}
```

Scopes

Scopes are also implemented using Contexts, e.g., in Listing 5.3, **play grouping** is a scope which is represented as a Context. To implement searching the parent scope, a reference to that scope is appended to the end of the Context. In this way, when searching for a name which is not found in a scope, the search will continue in a parent scope. An illustration of this idea can be seen in Listing 5.4, which is an implementation of the THAPL presentation shown above with the enclosing scope reference included. The angle brackets (**<global context>**) represent a reference.

Listing 5.4: Context construction for “play grouping” with link to enclosing scope

```
{
    "play grouping": {
        "group": {
            "first": { ..., <"group" context> },
            "second": { ..., <"group" context> },
            "third": { ..., <"group" context> },
        },
        <GLOBAL context>,
    },
}
```

Inheritance

Inheritance is accomplished in much the same way as scopes, i.e., by including a reference to the inherited entity in the Context of the inheriting entity. This provides a straight-forward way to implement inheritance semantics, such as overriding, shadowing, and so on. Similarly, multiple inheritance is implemented by appending multiple Context references to the inheriting entity’s Context; this results in multiple-inheritance semantics that are similar to C++. The full representation of Contexts in **play grouping** is shown in Listing 5.5.

Tagging

When searching inheritance trees, the third rule for name resolution (see Section 5.2) requires “stickiness” when searching inheritance trees; i.e., when searching the inheritance tree for a name, the search will not proceed to the scope of a parent entity.
Listing 5.5: Context construction for “play grouping” with inheritance and scopes
{
  "play grouping": {
    "group": {
      "first": { "<node" context>, "<group" context> },
      "second": { "<node" context>, "<group" context> },
      "third": { "<node" context>, "<group" context> },
    },
    "GLOBAL context",
  },
}

As there is no differentiation between a parent scope reference and a parent entity reference, the mechanisms described above are not sufficient to meet this requirement. Therefore, THAPL adds a reference wrapper, that tags the reference with this additional metadata, and allows pruning of the search tree according to THAPL’s name resolution rules.

Other Values

Previously it was shown THAPL uses the Context data-structure to implement entities and scopes, but the same data-structure is also utilized for other aspects of the language. First, in order to implement can properties and the divisional structures, directives can be attached to Context objects. Second, when calling can properties and divisional structures, the current scope is a dynamically constructed Context representing the calling scope; this is a form of dynamic scoping. Finally, keys in Context objects can refer to primitive values, of the types specified in Section 4.3.1.

5.2.2 Simulating Other Languages

While the Context data-structure is primarily used to implement THAPL, it can also be used to implement the semantics of other languages, with the appropriate modification and convenience methods. For instance, the semantics and class hierarchy of Little Smalltalk [Budd, 1987], ignoring matters of syntax, can be implemented using the Context data-structure.

Specifically, the representation of a Little Smalltalk object will be similar to that used in THAPL, but with minor changes to accommodate the fact that Little Smalltalk is a class-based language, as opposed to THAPL which is prototype-based. As can be expected, a class will be represented using a Context. This Context will then contain, in order, keys for all the messages that instances of this class can respond to, keys for each instance variable, and (optionally) a Context linking this class to its super-class.

Instances are also represented using Contexts. However, instances do not have any messages their instances respond to, as they don’t have instances of their own, not being classes. Nonetheless, in this implementation instances respond to the “class” message,
returning their class; this corresponds to the class pointer described in the original implementation. The Context objects representing messages link to an objects which contain the method’s temporary variables, as well as the “bytecode” for executing the method, in a similar fashion to the way THAPL’s commands are contained in Context objects.

From this description, the operation of the object description becomes clear: to send a message to an instance, its class entry is consulted and the message name is resolved in that Context. Creating new instances (using the class’s `new`) simply involves creating a new Context object, setting the class entry, and calling the constructor.

See Appendix A.2 for a partial Python implementation of Little Smalltalk using the Context data-structure.

## 5.3 Animation Semantics

The exact computations used by THAPL for creating animations are described in Chapter 6. In this section, a brief overview of the semantics of animations in the language is provided. The method by which the various component parts of the animation are rendered into their final form is also provided.

### 5.3.1 Primitives

The primitive that creates an atomic animation in THAPL is the `change` atomic directive, defined in Section 4.4.2. Each `change` directive is an instruction to the rendering engine to transform a value over time, which creates an animation. For discussion purposes, we will use “length” to refer to the length of the animation in slides, and “magnitude” will be used to account for the size of the difference between the old and new values. For instance, if `x` is an integer that is initialized to 1, `change x to 4` will, unsurprisingly, change `x` to 4; but it will do this by generating an animation that is 3 slides long. This is a result of THAPL’s default behavior, wherein the length is equal to the magnitude; but this can be controlled by adding parameters to the directives. For instance, specifying `change x to 4 in 6 slides` will create a slower animation.

Note however that the length specified in the `change` directive is a hint, and the compiler may choose another value if forced to do so by another constraint in a compound animation. The `change` directive can also specify the `k` constant of this animation, a constant factor that allows for prioritization when multiplexing several animations, e.g., `change x to 4 in 6 slides with k of 3`. By default, `k = 1`. A higher value for `k` instructs THAPL to treat this constraint as more important, and to keep the length of the animation closer to the number given in the `change` directive.

The exact system for calculating the length of animations, including the precise way `k` is used, is described in Chapter 6.
5.3.2 Solving for Slides

In Chapter 6, the mechanism that decides the length of each part of the animation is explored. However, even after the compiler calculates the length of each animation, several issues remain:

1. Atomic animations need to be stretched or compressed to their new length by some method,
2. Animation lengths, as calculated by springs, are in \( \mathbb{R} \), but slides are atomic and cannot be subdivided,
3. The entities must be rendered into a slide.

Mutations

Mutations describe atomic animations in ThapL, by specifying the change from one value to another between two points in time. The way a mutation operates on a value depends on that value’s type. Two types of mutations are currently implemented:

- **Numeric mutations**, which modify an integer or real value progressively,
- **Instant mutations**, which change a field from its old value to new value at a specific instance, typically at the half-way point of the animation.

Numeric mutations operate in the following manner: The rate of change starts at zero, then slowly increases to a constant speed, and finally slows down to zero. This can be seen in Fig. 5.1, which depicts a graph of speed vs. time. Note how the area under the curve represents the difference between the old value and the new one. In user-experience (UX) terminology, this is called an “ease-in-out” curve.

Discretizing Slides

As mentioned, another challenge is that of discretizing animations into slides, the atomic unit of ThapL. This is performed in the following way:
1. **Thapl directives** are compiled into an intermediary form, called **instructions**.

2. Instructions are performed, resulting in a **log**, which contains each animation, and its start and ending times.

3. Each log is split into two **events**, representing the **start** and **end** time of the log. These events are put into the **event list**.

4. **Key events** are added to the event list; these events have an integer time, one for each slide in the animation. The number of key events is decided by the maximum end time from the log.

5. The event list is sorted according to time.

6. Holding an active log list, the event list is iterated over. For each event, one the following actions are performed:
   - If it is a start event, the associated log is added to the active log list.
   - If it is a end event, the associated log is removed from the active log list.
   - If it is a key event, the associated field for each log entry in the active log list is updated the associated field for each log by computing the value at the time of the key event using the mutation. Finally, a slide is emitted.

   This method successfully discretizes the animations into indexed slides, ensuring that all frames are accounted for, especially the first and final frames which are the most important. This is critical, since if an animation is calculated to be 1.1 slides long, it should not be truncated and left in a state where the final slide does not show its specified end-state.

**Rendering**

To render a slide, the actors are iterated over, and each one is rendered. Each type is associated with a renderer that outputs a suitable value; e.g., the integer renderer prints an integer, the string renderer outputs the string, and so on. The exception is the **Compound** renderer, which either outputs nothing if the entity does not inherit from **animable**, and the contents of the **render** property if it does.

This process is then repeated for every slide in the animation, creating the final output of the Thapl program.
Chapter 6

Physical Constraint Model: Springs

This chapter explains how THAPL creates slides from sequential and parallel actions using springs. This ability to declaratively schedule animations is THAPL’s main contribution to the art. As we previously saw in Section 4.4.1, there are two ways to compose THAPL directives: in sequence using THEN, and in parallel using MEANWHILE.

For instance, consider the following THAPL compound directive:

```
message send THEN (
    (robot arm throw THEN robot arm retract)
    MEANWHILE
    robot light blink)
```

In this example, the robot arm should start throwing after the animation for sending a message has completed. Note that the directive does not contain any slide numbers for when each part of the animation happens; that is left to THAPL to calculate. We require a method to decide the length of each animation, as well as its start time. That is, for each directive seen in the example, we need to know when, in term of slide numbers, the animation begins, and when it ends. This is obviously predicated by the structure imposed by the composition; in THAPL, in order to be able to control the animation, we decided that composition of directives should obey the following constraints:

1. Directives composed using MEANWHILE start together and end at the same time;
2. Directives composed using THEN occur in sequence, i.e., the end time of the first directive is the start time of the second directive.

In addition, we would like our model to permit specifying the natural length of the animation, i.e. the length it would be if it is not constrained by composition with another directive in parallel, and a tweakable value that provides some control over how time is distributed across the different animations.

The model used by THAPL to meet these requirements is inspired by the physical model of Hookean springs [Hooke, 1678]. In this model, springs have two properties:
a natural, non-deformed length $\ell$, and a characteristic factor, or stiffness, $k$. Linear springs then obey Hooke’s law: $\vec{F} = k \cdot \Delta \ell$; that is, the force that is required to deform the springs by $\Delta \ell$ is $k \cdot \Delta \ell$. To use this model, we attach a spring to each directive, and during composition we connect the springs in parallel for parallel composition, and in sequence in the case of sequential composition, resulting in a system of springs. We can then solve this system to obtain the length of each spring in the system, which represents the length of each directive.

As an example, let us again consider the example directive from earlier. In Fig. 6.1 we see the syntax tree that is created from this directive. Each internal node represents composition in a compound directive, and each leaf node is an atomic directive. We attach a spring to each leaf node, then we can create a spring system in each internal node, connecting the springs from its children in sequence or in parallel, as required. In Fig. 6.2 we can see the spring system that is constructed from this compound directive.

The length of the spring system corresponds to the time of the entire compound directive, and the length of each spring in the system corresponds to the time of each atomic directive.

We now continue to a more in-depth discussion of the calculations required.
6.1 The Physical Model

This section is divided into two parts. First, we discuss the simple, high-school physics model of linear springs and equivalent springs, where \( k \) is a constant function. Then, we extend the model into discrete non-linear springs, by making \( k \) a step function.

6.1.1 Linear springs

Deriving Equivalent Spring Formulas

As mentioned earlier, the fundamental property of springs is that they obey Hooke’s law:

\[
\vec{F} = k \cdot \Delta \ell
\]  

(6.1)

It is from this property that we can derive the formulas for composing springs. First, all of our calculations are performed in a single dimensions, therefore we treat \( \vec{F} \), the vectorial force the spring applies or its tension, as the scalar \( F \).

**Springs in series** When we connect springs in series, the displacement of the entire system is equal to the sum of the displacement of each spring:

\[
\Delta \ell_{\text{total}} = \sum_i \Delta \ell_i
\]  

(6.2)

Additionally, the tension is equal between each spring; in other words, the tension in each spring is equal to the tension in the system:

\[
F_i \equiv F
\]  

(6.3)

To find the spring constant of a virtual spring that is equivalent to the entire system, we manipulate Eq. (6.1) to give the displacement as a function of the tension \( F \):

\[
\Delta \ell = \frac{1}{k} \cdot F
\]  

(6.4)

We then sum over all the springs to find the total displacement:

\[
\Delta \ell_{\text{total}} = \sum_i \Delta \ell_i = \sum_i \left( F_i \cdot \frac{1}{k} \right) = \left( \sum_i \frac{1}{k_i} \right) F
\]  

(6.5)

By comparing Eq. (6.5) to Eq. (6.4), we see that the equivalent spring constant \( k_{\text{eq}} \) for springs connected in series is:

\[
\frac{1}{k_{\text{eq}}} = \sum_i \frac{1}{k_i}
\]  

(6.6)

So far we’ve discussed the displacement \( \Delta \ell \), but we’d like to know the natural length \( \ell_{\text{eq}} \), or the length of the spring when no forces are applied to it, of the equivalent spring.
as well. In the series case, it is equal to the sum of the natural length of the comprising springs:

$$\ell_{eq} = \sum \ell_i$$  \hspace{1cm} (6.7)

**Springs in parallel** The derivation of the parallel case diverges from the high-school physics derivation, which usually assumes that the natural length is the same for all springs. This is because our model requires springs in parallel to have the same length, even at rest. In Fig. 6.2, if the parallel springs were to have different natural lengths, we would expect the plates to rotate, and stop being parallel. Instead, in our model, the mathematical springs are all connected to the same two points in a single dimension. Intuitively, we can imagine this as an assembly where the parallel springs are connected to plates, as in Fig. 6.2, but the plates are on frictionless rails that force them to remain parallel.

Therefore, we must first derive the natural length of the equivalent spring $\ell_{eq}$. The total tension in the system is equal to the sum of each spring’s tension:

$$F = \sum_i F_i$$  \hspace{1cm} (6.8)

When the spring system is at its natural length then $F = 0$, therefore by Eq. (6.1):

$$0 = \sum_i F_i = \sum_i k_i \cdot \Delta \ell_i$$  \hspace{1cm} (6.9)

We know that the length of the system is equal to the length of a single spring within it, which is its natural length plus the displacement:

$$\ell_{eq} \equiv \ell_i + \Delta \ell_i$$  \hspace{1cm} (6.10)

By manipulation of Eqs. (6.9) and (6.10) we have that:

$$0 = \sum_i k_i \cdot (\ell_{eq} - \ell_i)$$  \hspace{1cm} (6.11)

Which by manipulation gives us the formula for the equivalent spring’s length:

$$\ell_{eq} = \frac{\sum_i k_i \cdot \ell_i}{\sum_i k_i}$$  \hspace{1cm} (6.12)

We can now derive the equivalent spring constant in the parallel case. We begin by noting that the displacement of each spring in the system is comprised of the displacement in the resting system $\Delta \ell_i^*$, and the additional displacement $\Delta \ell_i'$:

$$\Delta \ell_i = \Delta \ell_i^* + \Delta \ell_i'$$  \hspace{1cm} (6.13)
The displacement of the entire system is equal to the additional displacement of each spring, since the springs are connected in parallel:

\[ \Delta \ell_{\text{total}} = \Delta \ell'_i \quad (6.14) \]

From here we apply Eqs. (6.1) and (6.11), to get:

\[ F = \sum_i k_i \cdot \Delta \ell_i = \left( \sum_i k_i \cdot \Delta \ell'_i \right) + \left( \sum_i k_i \right) \cdot \Delta \ell_{\text{total}} = \left( \sum_i k_i \right) \cdot \Delta \ell_{\text{total}} \quad (6.15) \]

Finally, by comparing to Eq. (6.1), we see that the equivalent spring constant is the sum of the comprising springs’ constants:

\[ k_{\text{eq}} = \sum_i k_i \quad (6.16) \]

**Partitioning equivalent springs**  In addition to composing springs into parallel and serial spring assemblies and calculating their equivalent properties, we would also like to be able to break apart these spring assemblies into separate springs. That is, given a spring assembly that is stretched to a certain length \( \ell_{\text{eq}} \), we would like to calculate the length \( \ell_i \) of each spring in the assembly. The parallel case is trivial:

\[ \ell_i = \ell_{\text{eq}} \quad (6.17) \]

In the series spring assembly case, from Eqs. (6.1) and (6.3) we have:

\[ k_i \cdot \Delta \ell_i = k_{\text{eq}} \cdot \Delta \ell_{\text{eq}} \quad (6.18) \]

Which we can then manipulate to get:

\[ \Delta \ell_i = \Delta \ell_{\text{eq}} \cdot \frac{k_{\text{eq}}}{k_i} \quad (6.19) \]

Note that this formula is in terms of \( \Delta \ell_i \) and \( \Delta \ell_{\text{eq}} \); the actual value of interest is \( \ell'_i \), the compressed length of the individual spring.

These formulae are summarized in Table 6.1.

### Table 6.1: Summary of linear spring formulas

<table>
<thead>
<tr>
<th></th>
<th>Parallel</th>
<th>Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>equivalent spring constant</td>
<td>( k_{\text{eq}} = \sum_i k_i )</td>
<td>( 1/k_{\text{eq}} = \sum_i 1/k_i )</td>
</tr>
<tr>
<td>equivalent natural length</td>
<td>( \ell_{\text{eq}} = \sum_i k_i \cdot \ell_i ) ( \ell_i = \ell_{\text{eq}} )</td>
<td>( \ell_{\text{eq}} = \sum_i \ell_i )</td>
</tr>
<tr>
<td>partitioning</td>
<td></td>
<td>( \Delta \ell_i = \Delta \ell_{\text{eq}} \cdot \frac{k_{\text{eq}}}{k_i} )</td>
</tr>
</tbody>
</table>
6.1.2 The spring group(s)

It is interesting to note that we can define two spring abelian groups $G_+$ and $G_\times$, which are comprised of the set of springs and a binary operation: $+$ which composes springs in sequence, and $\times$ which composes them in parallel. This is required for correctness in this application; we would like the composition of springs to lead to the same result regardless of the ordering the composition was processed in. It is clear that when connecting two springs, $A$ and $B$, in series, then the order in which they appear matters; if $A$ is first, then we want the animation attached to $A$ to occur first. However, the equivalent spring created by the composition of $A$ and $B$ should be the same regardless of the internal order of $A$ and $B$. Similar concerns lead to the desire for associativity when composing springs.

In order to define the operators such that they are associative, we must tweak the previously defined formulae. The first modification required is the definition of the concept of compliance, $c = 1/k$.

Given three springs, $A$, $B$, and $C$ with spring constants $k_A$, $k_B$, and $k_C$ respectively, and natural lengths $\ell_A$, $\ell_B$, and $\ell_C$ respectively, then:

**Definition 6.1.1.** The binary operation $+$ composes springs in series. $A + B$ is an equivalent series spring, such that $c_{A+B} = c_A + c_B$ and $\ell_{A+B} = \ell_A + \ell_B$.

**Claim 6.1.2.** $G_+$ is an abelian group of springs under $+$.

**Proof.** Let $A, B, C \in G_+$ be springs. We show that $G_+$ has the following properties:

1. Associativity:

   \[
   c_{(A+B)+C} = c_A + c_B + c_C \\
   = c_A + c_B + c_C \\
   = c_{A+(B+C)} \\
   = c_{A+B+C}
   \]

   \[
   \ell_{(A+B)+C} = \ell_A + \ell_B + \ell_C \\
   = \ell_A + \ell_B + \ell_C \\
   = \ell_{A+(B+C)} \\
   = \ell_{A+B+C}
   \]

   \[
   (6.20)
   \]

2. Commutativity:

   \[
   c_{A+B} = c_A + c_B \\
   = c_B + c_A \\
   = c_{B+A}
   \]

   \[
   \ell_{A+B} = \ell_A + \ell_B \\
   = \ell_B + \ell_A
   \]

   \[
   (6.22)
   \]

56
3. Identity: The zero element is the zero spring, such that \( c_0 = 0 \) (i.e., \( k_0 = \infty \)) and \( \ell_0 = 0 \).

4. Inverse: Each spring has an inverse spring with an inverse natural length and compliance. This is not actually useful in our model, but it is required for the proof.

Therefore, \( G_+ \) is an abelian group of springs under \(+\) by definition.  

**Definition 6.1.3.** The binary operation \( \times \) composes springs in parallel. \( A \times B \) is an equivalent parallel spring, such that \( k_{A \times B} = k_A + k_B \) and \( \ell_{A \times B} = \frac{k_A \cdot \ell_A + k_B \cdot \ell_B}{k_{eq}} = k_A \cdot \ell_A \cdot c_{eq} + k_B \cdot \ell_B \cdot c_{eq} \).

**Claim 6.1.4.** \( G_\times \) is an abelian group of springs under \( \times \).

Here we relax rigor slightly by manipulating the definition of equivalent natural length to be \( \ell_{eq} = \sum_i \ell_i \cdot k_i \cdot c_{eq} = \sum_i m_i \), and treating \( m_i \) as a constant. We can implement this by deferring evaluating \( \ell \), and therefore \( m_i \), to after the equivalent spring has been constructed; the proof requires this relaxation due to the multiplications within \( m_i \).

**Proof.** Let \( Aq, B, C \in G_\times \) be springs. We show that \( G_\times \) has the following properties:

1. **Associativity:**

\[
k_{(A \times B) \times C} = \frac{k_A \cdot \ell_A + k_B \cdot \ell_B}{k_{eq}} = k_A \cdot \ell_A \cdot c_{eq} + k_B \cdot \ell_B \cdot c_{eq} = \frac{k_A \cdot \ell_A + k_B \cdot \ell_B}{k_{eq}}
\]

2. **Commutativity:**

\[
k_{A \times B} = k_A + k_B = k_B + k_A \quad (6.26)
\]

3. **Inverse:** As previously, each spring has an inverse with a negated natural length and stiffness constant. This inverse is actually not needed in practice.

57
4. Identity: The identity element is the unit spring, such that \( k_u = 0 \) (i.e., \( c_u = \infty \)) and \( \ell_u = 0 \).

Therefore, \( G_x \) is an abelian group of springs under \( \times \) by definition.

The implementation performs the composition of springs in a similar manner to the proofs shown here.

### 6.1.3 Discrete non-linear springs

In the previous sections we discussed linear springs, where \( k \) is a constant. However, in THAPL we would like to have springs with variable stiffness for stretching and compressing. For example, a directive that has a minimum length, but can otherwise stretch with spring constant \( g \). To specify this, we would like the spring attached to that directive to have an infinite stiffness parameter when compressing, but a stiffness parameter of \( g \) when stretching. We can define this behavior as a step function:

\[
k(x) = \begin{cases} 
\infty & \text{if } x \leq \ell, \\
g & \text{if } x > \ell
\end{cases}
\]

As the sum and product of any two step functions is another step function (with more steps), these functions compose as required by the groups previously defined.

#### Changes from the linear model

All of the operations remain identical to the linear model (see Table 6.1), save for one: The formula for equivalent natural length in the parallel case is modified to:

\[
\ell_{eq} = \frac{\sum_i \ell_i \cdot \ell_{eq}}{\ell_{eq}(\ell_{eq})}
\]  

(6.28)

Since \( \ell_{eq} \) is now defined as a linear recurrence, we require an iterative approach: we guess an initial \( \ell_{eq} \) (possibly the mean natural length), then iterate until reaching a fixed-point. However, we must first guarantee that a fixed-point will be reached.

For example, consider the following case, where \( k_2 \) is a step function with three steps (possible due to spring composition):

\[
\ell_1 = 4, k_1(x) = \begin{cases} 
5 & \text{if } x \leq 4, \\
10 & \text{otherwise}; \\
60 & \text{if } x \leq 2, \\
1 & \text{if } x \leq 3.5, \\
20 & \text{otherwise};
\end{cases}
\]

\[
\ell_2 = 2, k_2(x) = \begin{cases} 
60 & \text{if } x \leq 2, \\
1 & \text{if } x \leq 3.5, \\
20 & \text{otherwise};
\end{cases}
\]

To calculate \( \ell_{eq} \), we start by guessing an initial \( \ell_{eq} = 3 \), then plugging into
Eq. (6.28):

\[
\begin{align*}
\ell_1 \cdot k_1(3) + \ell_2 \cdot k_2(3) &= \frac{4 \cdot 5 + 2 \cdot 1}{5 + 1} = 3.6 \\
4k_1(3.6) + 2k_2(3.6) &= \frac{4 \cdot 5 + 2 \cdot 20}{5 + 20} = 2.4 \\
4k_1(2.4) + 2k_2(2.4) &= \frac{4 \cdot 5 + 2 \cdot 1}{5 + 1} = \frac{2}{3}
\end{align*}
\]

The formula will now oscillate between 2.4 and \(3^2/3\), never reaching a single solution. This is a problem for THAPL, since we require a well-defined length for every spring in order to be able to calculate the duration of the animation.

The solution is to limit the choice of \(k\) such that we guarantee a single solution. Requiring all the functions to be monotonic in the same direction has this desirable property; in THAPL, we chose to have all functions monotonically non-increasing. This allows encoding a directive with a minimum length that can stretch, which is an important tool in the language.

**Claim 6.1.5.** If all \(k(x)\) in a spring assembly are non-negative step functions with a finite number of steps that are monotonic in a shared direction and all \(\ell\) are non-negative, then it is guaranteed that we will reach a fixed-point.

**Proof.** Assume without loss of generality that every \(k(x)\) is a monotonic non-increasing step-function, that is, for all \(x, x' \in \mathbb{R}\) such that \(x < x'\) then \(k(x) \geq k(x')\).

Consider Eq. (6.28). By evaluating the difference between \(\ell_{eq}(x)\) and \(\ell_{eq}(x')\), we can see that the difference in the numerator is:

\[
\sum_i \ell_i \cdot (k_i(x) - k_i(x'))
\]

and the difference in the denominator is:

\[
\sum_i (k_i(x) - k_i(x'))
\]

since for all \(i\), \(\ell_i \geq 0\) and \(k_i(x) \geq k_i(x')\), it holds that the difference in the numerator is larger than the difference in the denominator. Therefore, \(\ell_{eq}(x) \geq \ell_{eq}(x')\), or in other words, the change in \(\ell_{eq}\) in each iteration will also be monotonic. There are a finite number of steps in each \(k_i(x)\), and therefore if the change is monotonic in the worst case we will eventually reach the last step in each \(k_i(x)\), and we are guaranteed to reach fixed-point. 

Therefore, in order to implement non-linear springs in THAPL we require the prerequisites listed in Claim 6.1.5.
Chapter 7

Results

This chapter is concerned with analyzing animations constructed using THAPL, especially in comparison with equivalent \LaTeX{} animations. We strive to show that THAPL meets the goals of producing animations in which the underlying code is:

1. Maintainable, i.e., simple to modify,

2. Readable, simple to understand, and descriptive; the code should be at a correct level of abstraction,

3. Flexible, meaning that it should be possible to create many different types of presentations.

7.1 Fitness For Purpose

In order to show that THAPL solves real-world problems, the \TeX{} – \LaTeX{} Stack Exchange question & answer website\(^1\) was searched for questions demonstrating desire to create animations. Then, an equivalent THAPL program was written for that that animation, so that \LaTeX{} and equivalent THAPL code can be compared. Since some of the listings are long, they are provided in full in Appendix B, while excerpts are displayed here.

7.1.1 Complex Animations

In “Make latex presentation with complex animation”\(^2\) user \texttt{sreraj t} wants to create an animation where various boxes containing an explanation, or tool-tips, show up in sequence, in order to explain the intricacies of a certain on-screen image. As a starting point, \texttt{user36296} provides the presentation seen in Listing 7.1. Note the usage of \texttt{visible on} in order to control the showing of tool-tips.

\footnotesize

\(^1\)https://tex.stackexchange.com/

\(^2\)https://tex.stackexchange.com/q/482587/28020
Listing 7.1: Stack Exchange answer for complex animation

\documentclass\{beamer\}
\usepackage\{tikz\}
\usetikzlibrary\{overlay-beamer-styles\}
\begin\{document\}
\begin\{frame\}
\begin\{tikzpicture\}
\node \{\includegraphics[width=.5\textwidth]{example-image}\};
\node[visible on=<2>] (a) at (4,-2) \{text\};
\draw[<-,red,visible on=<2>] (0,1) -- (a);
\end\{tikzpicture\}
\end\{frame\}
\end\{document\}

Using \textsc{Thapl}, we can specify this behavior easily using \texttt{appear} to display the tool-tips, as seen in Listing 7.2. Note that we no longer have to explicitly reference the slide number. This is one of the simpler examples, but it is listed here as an important use-case. Also interesting to note is the light wrapping \textsc{Thapl} provides around \LaTeX, allowing the use of naked \LaTeX code without abstraction. The full capabilities of the underlying drawing framework, TikZ, are available, and the language does not limit what we are able to draw on-screen. This is true regardless of the underlying language, so no capabilities are restricted by using \textsc{Thapl} as a pre-processor.

Listing 7.2: \textsc{Thapl} version of “complex animation”

\begin{verbatim}
PLAY hanoi
LOAD library\animable, library\location from ".\library.thapl"

ACTORS
  image (animable "\node \{\includegraphics[width=.5\textwidth]{example-image}\};")
  tooltip a:
    node (animable "\node (a) at (4, -2) \{text\};" BUT visible = false)
  line (animable "\draw[<-,red] (0,1) -- (a);" BUT visible = false)

ACTION
  tooltip a appear
\end{verbatim}

7.1.2 Simulating Logic Circuits

In “How do I simulate a logic circuit in \LaTeX?”, user \texttt{afsara\_ben}\textsuperscript{3} is attempting to depict a simulation of a flip-flop circuit. \textit{J Leon V.} answers with an example drawing

\texttt{https://tex.stackexchange.com/q/440287/28020}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{logic_circuit.pdf}
\caption{Logic Circuit}
\end{figure}

\textsuperscript{3}https://tex.stackexchange.com/q/440287/28020
the circuit and controlling the behavior of its simulation with a large for-loop, seen in Listing 7.3 (the full listing can be found in Appendix B.2.1.) This is conceptually sim-

ilar to the way THAPL controls the behavior of entities in a separate location from the description of those elements. However, the syntax requires keeping track of the meaning of each column (this could be made simpler with the judicious use of indentation,) and requires the entire state of the circuit to be repeated for each line. In THAPL, we would instead describe the changes that occur in the graphic; in the equivalent THAPL implementation of this presentation (full version in Appendix B.2.2), we can see that although the the definitions part of the presentation is quite large, the ACTION part, shown in Listing 7.4, is compact, stemming from the fact that THAPL keeps track of the state of the entities.

Listing 7.4: THAPL implementation of a circuit simulation presentation (excerpt)

```
\foreach \C/\S/\R/\NR/\Q/\NQ/\STAT/\vC/\vS/\vR/\vNS/\vNR/\vQ/\vNQ in {
  LOW/LOW/HIGH/HIGH/NC/NC/No Change/0/0/1/1/Q_0/\OVERLINE(Q_0),
  LOW/LOW/HIGH/HIGH/NC/NC/No Change/0/0/1/1/Q_0/\OVERLINE(Q_0),
  LOW/LOW/HIGH/HIGH/NC/NC/No Change/0/0/1/1/Q_0/\OVERLINE(Q_0),
  LOW/LOW/HIGH/HIGH/NC/NC/No Change/0/0/1/1/Q_0/\OVERLINE(Q_0),
  LOW/LOW/HIGH/HIGH/NC/NC/No Change/0/0/1/1/Q_0/\OVERLINE(Q_0),
  LOW/LOW/HIGH/HIGH/NC/NC/No Change/0/0/1/1/Q_0/\OVERLINE(Q_0),
  HIGH/LOW/HIGH/HIGH/LOW/RESET/SET/Reset/1/0/1/1/0/0/1,    
  HIGH/LOW/HIGH/HIGH/LOW/RESET/SET/Reset/1/0/1/1/0/0/1,    
  HIGH/LOW/HIGH/HIGH/LOW/RESET/SET/Reset/1/0/1/1/0/0/1,    
  HIGH/LOW/HIGH/HIGH/LOW/RESET/SET/Reset/1/0/1/1/0/0/1,    
  HIGH/LOW/HIGH/HIGH/LOW/INVALID/INVALID/Invalid/1/0/0/0/0/Q/Q',
  HIGH/LOW/HIGH/HIGH/LOW/INVALID/INVALID/Invalid/1/0/0/0/0/Q/Q'){
```

7.1.3 Hannukah Menorah

![Hannukah Menorah](https://tex.stackexchange.com/q/85844/28020)

Figure 7.2: Hannukah Menorah

In “How can we draw a Hannukah Menorah with decorations, using TikZ?”[^4], user

[^4]: https://tex.stackexchange.com/q/85844/28020
Raphael expresses an interest in drawing a Hannukah Menorah (חנוכיה) using TikZ. User Paul Gaborit answers with a simple black & white illustration, and also shows how a simple animation showing the progression of days can be created based on the illustration, seen in Listing 7.5. This example is, while quite short, is still complex enough that understanding how it works is non-trivial. In Appendix B.3.1, we show an equivalent Thapl program, that has some minor enhancements over the version shown in Listing 7.5: The Shamash has a customizable height difference from the rest of the candles, and the animation shows how the Menorah is to be lit on each day. A still-frame from the animation is seen in Fig. 7.2. In Listing 7.6 we see the array definition for the candles (actually the flame part of the candle), which allows us to succinctly describe and control solenoids on-screen items. Listing 7.7 displays part of the relevant action section, demonstrating the fine control over the individual flames offered by Thapl. To change the behavior of the animation, all that is required is rearranging the action directives.

Listing 7.5: Hannukah Menorah TikZ animation, from Stack Exchange

```
\documentclass[margin=2mm,tikz]{standalone}
\usetikzlibrary{arrows,shapes,positioning,calc}
\def\arraycandles{{0,4,3,2,1,-1,-2,-3,-4}}
\def\candlepath{
  to[out=0,in=-45] ++(0,5mm) to[out=-90,in=180] ++(0,-5mm)
  to[out=20,in=-45] ++(.3mm,2mm) to[out=-90,in=160] ++(-.3mm,-2mm)
}
\begin{document}
\foreach \day in {1,...,8}
\begin{tikzpicture}[line width=2mm]
  \foreach \pos in {1,...,4}{
    \draw (0:\pos * 6mm) arc(0:-180:\pos * 6mm);
  }
  \foreach \pos in {-4,...,4}{
    \fill (\pos * 6mm,2.5mm) ++(1mm,0) -- ++(1mm,2mm) -- ++(-4mm,0) -- ++(1mm,-2mm) -- cycle
    (\pos * 6mm,2.5mm) ++(1mm,0) -- ++(1mm,2mm) -- ++(-4mm,0) -- ++(1mm,-2mm) -- cycle;
  }
  \draw (0,0) -- (0,4 * -6mm - 6mm);
  \fill (0,4 * -6mm - 5mm) arc(0:-180:4mm);
  \foreach \candlenum in {0,...,\day}{
    \pgfmathtruncatemacro{\pos}{\arraycandles[\candlenum]}
    \draw[\pos * 6mm,5mm]\candlepath;
  }
\end{tikzpicture}
\end{document}
```

Listing 7.6: Hannukah Menorah: Thapl Entity definitions (excerpt)

```
candles[-4 .. -1, 1 .. 4] (candle \texttt{but} visible = false):
  \texttt{pos} = \((i)\)
shamash (candle \texttt{but} height = 5):
  \texttt{pos} = 0
```
7.2 Flexibility

To show that \textsc{Thapl} is a \textit{flexible} language, more presentations are required. In this section, three more \textsc{Thapl} programs are described to show a range of animations that are possible using the language.

\begin{verbatim}
HAS pos = 0
ACTION
  candles\item 4 appear
  candles disappear
  candles\item 3 appear \textit{THEN} candles\item 4 appear
  candles disappear
  candles\item 2 appear \textit{THEN} candles\item 3 appear \textit{THEN} candles\item 4 appear
  candles disappear
  candles\item 1 appear \textit{THEN} candles\item 2 appear \textit{THEN} candles\item 3 appear
\end{verbatim}

\textbf{Towers of Hanoi} The Towers of Hanoi animation that served as the focus point of Chapter 3 is a good example of a \textsc{Thapl} program. The full code of that program is included in Appendix B.1.2. The Towers of Hanoi presentation includes elements that show the strengths of \textsc{Thapl}, for instance having two elements moving at the same time. Additionally, modifying the presentation is straight forward. For example, adding more discs or poles is a straight-forward task, while in the original Ti\textsc{k}Z version it would be much more involved. The first four frames of the animation are shown in Fig. 7.3.

\textbf{Convolution} Another example of a \textsc{Thapl} presentation is the convolution animation, the code of which can be seen in Appendix B.4.1, frames from which can be seen in Fig. 7.4. The implementation style of this presentation is interesting: There’s a repeating “beat” to the presentation, $g \tau \text{ a u s t e p}$, which causes the upper graph to increase by 1, and the lower graph is controlled in parallel using the \texttt{MEANWHILE} composition operator. The correctness of the animation requires fine control over the progression of
the two graphs. This presentation is an example of how \textsc{Thapl} succeeds in providing the needed flexibility to make animations such as this.

![Graphs](image)

Figure 7.4: Four frames from a presentation explaining the convolution operation

**Intersection**  In Section 1.4 a simple example of a \textsc{Thapl} presentation was introduce (Listing 1.2). Some frames from that presentation can be viewed Fig. 7.5. While the action part of the animation is not complex, the example shows that making small a animation with just a couple of moving elements is straight-forward using \textsc{Thapl}.

### 7.3 Analysis

Throughout this document we have seen several examples of \textsc{Thapl} code. In addition to the examples shown in Section 7.1 and Section 7.2, Listing 1.2 showed a simple animation that moves circles around the screen, and Chapter 3 constructed an animation solving the Towers of Hanoi puzzle from scratch. While this is by no means a comprehensive dataset, it does allow us to draw conclusions on \textsc{Thapl}'s success in meeting its stated goals.

First, it is important to note that \textsc{Thapl} does not offer any particular support in drawing figures, which in the provided examples was performed using \textsc{TikZ}; as a matter of fact, in designing \textsc{Thapl} one of the major points of concern was avoiding inventing “yet another” figure drawing language. As a result of this decision, in most of the \textsc{Thapl} programs that we have seen, the entity definition section is much larger in size than the action specification section, due to the size of the \textsc{TikZ} drawing code. While this creates somewhat lop-sided \textsc{Thapl} programs, this is not actually a problematic
element of the language; in fact, it leads us to a major strength of the language, the separation of drawing code and animation code.

Separating these two disparate concerns makes sense, since when creating animations one usually needs to either modify how a thing looks or how it behaves in the animation. We argue that tweaking both at the same time is somewhat rare, and if needed can be done in an iterative manner. The separation allows the user to concentrate on one task, and improves the readability of each separated part by removing superfluous elements.

Another way in which Thapl creates a more maintainable language is the interaction model between animation steps, wherein a spring system is created from the declarative action section, which can then be evaluated and solved. The solution is then used to compile the final rendered animation. This computational model provides a way to modify existing animations, which can then self-correct in order to produce a final animation that matches the intention of the user without manually adjusting parts of the document that were not changed.

In summary, the examples we have seen show how Thapl meets its design goals of being a succinct, readable, and powerful domain-specific programming language for specifying behavior in slide-show based animations.
Chapter 8

Conclusion and Future Work

THAPL—the theatrical programming language—is a text-based domain-specific programming language (DSL), designed to tackle the challenge of specifying the behavior of on-screen objects in a readable, succinct manner, with the goal of making slide-show animations maintainable. THAPL’s design allows relegating the duty of rendering drawings to a different program, making it possible to to limit the scope of the language to handling behavior, i.e., the action that occurs during the animation, rather than handling the difficult-in-its-own-right task of encoding illustrations. The main argument put forth by THAPL is that focusing on specifying behavior, using approaches from the declarative programming paradigm, enables presentation authors to create considerably easier-to-maintain presentations.

The major inspiration for THAPL’s unique and open syntax are Shakespearean stage plays. THAPL attempts to emulate the basic feel and structure of written plays in order to better serve its intended purpose.

THAPL contains novel features and building blocks which may contribute to other programming languages and applications. These include:

- The context data-structure, an augmented dictionary, which was applied to implement the semantics of the THAPL language as described in Chapter 5;

- A declarative approach to defining relations between different parts of an animated slide-show, using a physics-inspired model based on springs, discussed in Chapter 6;

- A flexible, user-extensible syntax which forms a constructed language resembling English, defined in Chapter 4.

As detailed in Chapter 7, the value of THAPL as a concept are demonstrated by users’ numerous requests for assistance in the \textsc{\LaTeX} Stack Exchange website. From users’ questions, it is evident that the solution proposed in THAPL can be a powerful tool to better solve users’ real-world problems. Moreover, THAPL enables the user to author presentations whose code exhibits improved readability and maintainability, increasing one’s productivity and the ability to re-use previous work.
8.1 Related Research

Chapter 2 presented the state-of-the-art in the domain of animation generation in the context of DSLs, in order to present the backdrop in which THAPL should be considered. Here, we extend the survey to other domains, unrelated to animation.

8.1.1 Musical Composition

Constraint programming has been influential in musical composition [Anders and Miranda, 2011], as the declarative and modular approach inherent to the programming paradigm is similar to the way music theory is traditionally expressed. Additionally, similarly to animations, musical composition is naturally concerned with the dimension of time and the synchronization of different elements (e.g., different instruments.) As in THAPL, this requires a model by which to generate constraints; unlike THAPL, musical composition has an existing, implicit model: Music theory. Computer-aided composition systems exist to facilitate the ease in which a composer can apply these domain-specific constraints [Talbot et al., 2017, McLean and Dean, 2018] to her composition.

However, the time-domain facilities of popular computer-assisted composition languages is limited. For instance, OPENMUSIC [Bresson et al., 2011, Bresson and Agon, 2008, Haddad, 2008] only has rudimentary support for time constraints, mostly discussing the graphical display of sheet notation so that various instances notations are synchronized. Operations on these sheet notation time constraint objects are described as an extension. The reasoning for this lack of constraint-based control of time seems to be that musical time is a fundamental concept in composition, and users demand strict control of the time-signature and beat of the composition.

Another DSL for audio programming is CHUCK [Wang, 2008, Kapur et al., 2015], termed “strongly-timed” by its authors. That is, the language provides the program with awareness of its position in time, and the ability to control the program’s progress by explicitly advancing time. Different program components share this logical notion of time, making it possible to synchronize parallel code through control of time advancement. This solution provides fine-grained control and allows precise reasoning of the temporal behavior of a program, but is harder to maintain, since modifying the program requires the programmer to consider the effect of the changes on the different parts of the program and manually correct them. Furthermore, reusing components that have been precisely synchronized to a different program is difficult.

Constraint Programming

Constraint Programming [Mayoh et al., 1994] is a programming paradigm in which relations between variables are declared in the form of constraints, i.e., as conditions of an optimization problem. This makes constraint programming part of the declarative
programming paradigm. The definition brings to mind logic programming languages, but sub-paradigms like Constraint Imperative Programming (CIP) have been recognized. CIP seeks to keep the idea of assignment inherent to imperative languages whilst adding explicitly declared constraints when that makes sense. Thapl takes part in the CIP tradition, but employs the constraints in a different manner: instead of specifying relations between variables, Thapl programs specify relations between statements, concerning the duration of the statements.

8.1.2 Name-binding

Part of the work performed in this research is definition of the Context data-structure, described in Chapter 5, which is used to implement name resolution in Thapl including binding and inheritance. This data-structure itself is based on the idea of association lists, which are used extensively in Lisp. However, inheritance and binding are implemented in a different manner in the Common Lisp Object System [Steele Jr., 1984, DeMichiel and Gabriel, 1987] and other variants of Lisp.

NaBL [Konat et al., 2013, Neron et al., 2015] (Naming Binding Language) is a DSL for specifying name binding and scope rules in other DSLs, which is integrated into a language workbench called Spoofax [Kats and Visser, 2010]. This is formalized in the form of a scope graph representing the naming structure of a program, and a resolution calculus which describes how to resolve references to declarations within a scope graph. This provides a framework that is similar to what is used in Thapl, with the scope graph taking the place of the Context. The Context is slightly more generalized than scope graphs due to the existence of an order in the children list of each node. Using this ordering, it is possible to implement language features such as multiple inheritance, which are not supported by NaBL.

8.1.3 Other DSLs

The design and specification of DSLs is an active area of research in the field of programming language theory. In this section, we summarize recent research in DSL design and implementations that are more loosely related to Thapl.

Penrose [Ye et al., 2017, Ni et al., 2017] is a system, implemented using DSLs, for rendering and displaying mathematical diagrams from plain-text descriptions in mathematical notation. Like Thapl, the goal is to create a visualization from an abstract description in the problem space, though the domain of that description is different.

FACT [Cauligi et al., 2019] is used to specify computations which are sensitive to timing side-channel attacks in a safe manner. Timing side-channel attacks exploit programs that leak information due to time variability of the program, which depends directly on the input. For example, a naïve string comparison function compares two strings character-by-character, stopping when it finds a mismatch. This is fine in most
applications, but if used for comparing passwords this function would allow an attacker to deduce which character in the input is incorrect, significantly reducing the number of passwords she needs to try. A safe string comparison function would perform the same number of comparison operations regardless of the input. Similar concerns exist about any code that implements cryptograhical algorithms. FACT contributes safe language constructs for sensitive computations that compile into constant-time LLVM bitcode, i.e. compiling branching code in such a manner that both branches take the same amount of time. This is similar to THAPL’s parallelization constructs, with the additional guarantee of safe constant-time behavior which is not provided by THAPL.

Algorithm visualization languages

As mentioned in Section 2.2, Karavirta, et al. [Karavirta et al., 2010] define a taxonomy of algorithm animation languages, and measure such languages via four main categories: visualization, dynamics, user interactions, and meta-language facilities. These categories are then further split into more specific sub-categories, each of which contains sub-categories that define the parameter to be tested. This section is concerned with evaluating THAPL using these criteria, as defined by the paper.

The first category, visualization, is concerned with language features that are used to “create static visualizations describing one state”. This criteria is not relevant to THAPL, since in the context of THAPL it actually examines the language responsible for rendering illustrations, i.e., \texttt{LATEX}/\texttt{TikZ}. For instance, the first sub-category is vocabulary, i.e., the type and amount of shapes supported by the language. THAPL does not actually support the drawing of any shape as part of the language itself (though \texttt{LATEX}/\texttt{TikZ}, which we used with THAPL throughout this document, supports many shapes). The next two sub-categories are positioning and style, which again, THAPL has no specific support for; however, once again, \texttt{LATEX}/\texttt{TikZ} has a comprehensive and intricate positioning and styling system, which can be used in conjunction with THAPL.

The next category is dynamics, or the features of the language that are used to modify the visualization, thereby creating the animation. The article starts off with high-level data-structure concept operations, e.g., pops and pushes on stacks, and other high-level operations on trees, arrays, and other common structures. Similarly to the case in visualization, THAPL doesn’t provide any such data-structures of these as part of the language itself—but these can clearly be implemented using the language, as we have seen in examples. The sequencing sub-category is the first category which can be used to evaluate THAPL. It contains two concerns: the first is granularity, by which the authors mean the ability to create animations in which parts are skippable, i.e., interactivity, which is not supported by THAPL, but due to a lack of implementation, rather than a design decision. The second concern is concurrency, meaning “whether the language supports definition of concurrent operations”, which THAPL provides extensive support for, using springs as the constraint model (c.f. Chapter 6). It is notable that
four of the nine languages evaluated by the paper do not support concurrency at all, and that the language that provides the most support for concurrency is SVG, which the paper defines as a general-purpose animation language rather than an algorithm animation language. The timing sub-category “examines the support for timing of the animation”. As mentioned, THAPL provides for this with the spring constraint model, specifically by providing the ability to control the natural length and spring constant of the constraint using the \texttt{CHANGE} directive. The languages examined in the paper that support concurrency all provide a mechanism to delay the start of an animation, and some can also specify the duration of an animation. SVG is noted to be more versatile in the timing mechanisms it supports, e.g., number of repeats. None of the languages evaluated support a constraint model that is comparable to THAPL’s. In fact, the languages support concurrency by doing away with abstraction, and providing direct control of the the start and end times of the animation. As the abstraction level is diminished, versatility improves; however, the loss of the abstraction complicates usage. A similar situation occurs with CSS’s animated properties, which were discussed in Section 2.1; here too the animations can be explicitly staggered, which provides versatility but is tedious to program.

\textit{Animation effects} reflects on “the various ways the graphical features of an object can be modified”. In a certain respect, THAPL is somewhat limited, only offering a transformation for numbers (integers or floats), and having everything else “pop” into place at a pre-defined time. However, the values that THAPL transforms control \LaTeX/Ti\LaTeX properties such as colors, fills, styles, and so on, making elaborate animation effects possible. The category measures which \textit{attributes} are modifiable, specifically \textit{style} (e.g., color, line style, etc.) and \textit{visibility} (i.e., whether objects can be shown and hidden), as well as \textit{transformations}, which are ways the geometry of the object can be modified, and are defined to include \textit{rotate}, \textit{scale}, and \textit{translate}. THAPL supports all of these operations; four of the languages evaluated by the paper do not (curiously, these are the same languages that do not support concurrency.)

The final sub-category, \textit{programming constructs}, is not a suitable criteria for THAPL as defined in the taxonomy, as it assumes a language employing the imperative programming paradigm paradigm. THAPL does have: declarations, expressions, assignment, types, sequentially executed statements and subroutines, but it does not support branching or loops. However, these concepts are insufficient to describe THAPL’s control flow; notably absent is declarative parallelism. The taxonomy also does not evaluate if the language is object-oriented or not. Interestingly, the other languages in the evaluation also do not fit well into these criteria. While seven of the nine languages evaluated use some form of sequential statements, six of them do not include expressions, assignment, or types: this suggests that it is incorrect to evaluate them using imperative programming criteria.

The \textit{user interaction} category, which measures “how the language supports interaction between the user and the animation”, is not applicable for THAPL as it is currently
implemented, since it is outside the scope defined for the language. The category includes control, or basis commands like starting or stopping the animation, responding, or language features interactive questions and responses to user actions, changing, or modifying the algorithm visualization according to input, and annotation, or adding explanatory comments to the slide-show during the presentation. The scope of the language does not cover anything that happens after the animation is created; as currently implemented, Thapl simply creates a PDF file and any further user interaction is handled by the PDF display software.

The meta-language category evaluates information about the language. Thapl supports comments, but sadly does not currently have any debug facilities as part of the language implementation. This is definitely a desirable feature in a language, and a future version of Thapl should definitely support debugging animations. Thapl is designed from the ground up to be extensible by adding new verbs (can properties) to characters and actors, and it provides a well-defined interface for extending the language. However, Thapl does not have specific support for localization, nor for metadata about the author or subject of the slide-show (note however that \LaTeX does allow setting PDF parameters for author and title). Karavitra et al. define import/export as interaction with packages like course management tools, automatic assessment systems, and so on; Thapl does not yet have built-in support for this. The other languages assessed fare similarly; only two languages have debug facilities, and only SVG has support for import/export (the paper does not describe how SVG meets this definition). A majority of the languages are extensible, with the XML-based languages having an advantage in this respect due to XML’s inherent extensibility.

To sum, Karavitra et al.’s taxonomy makes some assumptions about algorithm animation languages that are irrelevant in the case of Thapl. Nonetheless, comparing Thapl to this set of categories does suggest avenues of improvement for the language, and underline the novelty in Thapl’s usage of declarative programming and its constraint-based concurrency system.

8.2 Future Directions

The Thapl prototype developed in the framework of this academic thesis is naturally limited in scope, and many features and tools that are important for specific use-cases have not yet been implemented. The following is an incomplete list of such items, which may be valuable additions to Thapl-like suites:

1. **Features:** adding to the functionality, flexibility and productivity of Thapl. Examples of such include:
   - Visualization of the stage as seen by a specific actor, i.e., as it “sees” the environment and other interacting entities;
• Dynamic script capability—changing the scenario in accordance with specific rules, or sampling from a specified range of script variables (e.g., Monte-Carlo simulation);

• Augmenting the behavior models of the actors, beyond the springs model described in Chapter 6.

• Adding more mutations (c.f. Section 5.3.2, providing an interface to customize atomic animations.

2. **Interfaces**: Contributing to, being contributed by, or merely working along in synchronization with existing applications. For instance:

- Sources of geographical visualization (e.g., Google Maps);
- Sources of actors properties and behaviour, e.g., creating an animation based on the output of another program;
- Back-end rendering using something other than \LaTeX, e.g., HTML5 and CSS3.

3. **Maturity**: The Thapl implementation is missing several quality-of-life features that can be helpful for development:

- Comprehensive standard library of templatable UI elements;
- Error-checking and recovery;
- An advanced type system;
- A debugger, and debugging facilities in general.

4. **Productivity and maintenance capabilities**: to better meet the needs of users and developers, including:

- User interface to support non-professional users (e.g., instructor in an arena training simulator, required to dynamically alter the scenario script for the trainees;
- Automatic reports, to support developers and debriefing;
- Automatically generated documentation.

### 8.3 Closing Notes

Thapl is an interesting dive into the field of programming language design, containing elements which have the potential to be of interest in other domains. The Thapl implementation itself consists of the compiler and several example presentations, some of which were shown in this document. The source code for the compiler as it was at the time of this Thesis’s submission can be found in Appendix A, and future updates will be posted online at [https://github.com/chaosite/thapl](https://github.com/chaosite/thapl).
Although THAPL’s origin is in the context of animated slide-shows, THAPL may have further application in other contexts, in which the basic paradigm of actors, stage and script is valid. Of specific interest among these may be the context of arena simulators, in which a complex arena of objects is simulated for the purpose of operational research, training, briefing and debriefing, and marketing (envisioning a systematic concept of operation to stake holders). Such use-cases may include, for instance, autonomous vehicles behaviour in complex urban environment, in which multiple actors share the same stage, acting and impacting one another in accordance with a script, as well as other civilian or even military tactical arena simulators, simulating the way humans, remotely operated vehicles, and autonomous entities interact in a specified arena. Such simulators require top-level animations of the arena, descriptive tools to develop and alter the scripts in which the entities interact, and powerful capabilities to debrief the actor interactions.

Other domains that can also benefit from THAPL’s approach to modelling behavior and interaction include UI/UX design, where THAPL can act as a description language to describe what happens on-screen when a user performs a certain action; testing languages, where complex scripts which include setting-up and bringing down server instances can be orchestrated (and parallelized) by THAPL; and music composition, especially for sample-based music.

To conclude, THAPL is a useful tool for declaratively generating slide-based animations. It simplifies the behavior specification aspect of the work, while retaining the full strength of the underlying drawing framework. It accomplishes this by implementing ideas from programming language design, specifically drawing inspiration from the declarative programming paradigm, and introducing novel ideas and features into the ostensibly imperative task of specifying behavior. Using THAPL an author can create animations which may be conveniently modified, fixed and iterated on over the life-time of the presentation.
Bibliography


[Lamport, 1994] Lamport, L. (1994). \textsc{L}\textsc{a}\textsc{t}e\textsc{x}. Addison-Wesley.


[Marcovitz, 2012] Marcovitz, D. M. (2012). Powerful PowerPoint for educators: using \textsc{V}isual \textsc{B}asic for \textsc{A}p\textsc{p}lications to make \textsc{P}ower\textsc{P}oint interactive. Libraries Unlimited.


[Rössling and Freisleben, 2001] Rössling, G. and Freisleben, B. (2001). \textsc{A}n\textsc{i}m\textsc{a}l\textsc{s}cript: An extensible scripting language for algorithm animation. In Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education, SIGCSE ’01, pages 70–74, New York, NY, USA. ACM.

[Rößling et al., 2004] Rößling, G., Gliesche, F., Jajeh, T., and Widjaja, T. (2004). Enhanced expressiveness in scripting using \textsc{A}n\textsc{i}m\textsc{a}l\textsc{s}cript2. In Proceedings of the Third Program Visualization Workshop, pages 10–17.


Appendix A

Code Listings

A.1 Thapl Compiler

A.1.1 Grammar

Listing A.1: thapl.lark: Grammar definition for Thapl

```
// Rules for basic structure of Thapl

?start: _NL* PLAY

PLAY: "play"i section_header _NL _INDENT (act|load_directive)* section_body
     (act|load_directive)* _INDENT section_footer

ACT: "act"i section_header _NL _INDENT (scene|load_directive)* section_body
     (scene|load_directive)* _INDENT section_footer

SCENE: "scene"i section_header _NL _INDENT load_directive* section_body load_directive*
     _INDENT section_footer

section_header: tokens_identifier
section_footer: _DEDENT

load_directive: "load"i load_list "from"i string _NL
load_list: tokens_identifier ("," tokens_identifier)*

section_body: [scenery] [dramatis_personae] [action]

dramatis_personae: actors_header _NL _INDENT thaplon _DEDENT
actors_header: _ACTORS

scenery: scenery_header _NL _INDENT thaplon _DEDENT
scenery_header: _CHARACTERS

ACTION: _SCRIPT _NL _INDENT directives _DEDENT

_ACTORS: ("actors"i"dramatis personae"i)
_CHARS: "characters"i
_SCRIPT: ("script"i"action"i)

// Rules for Thapl directives

_directives: [_NL] compound_directive (_NL compound_directive)* [_NL]
```

83
compound_directive:  compound_directive THEN atomic_directive -> then_directive
| compound_directive MEANWHILE atomic_directive -> meanwhileDirective
| atomic_directive -> single_compound_directive

atomic_directive: CALL tokens_identifier -> call_directive
| RELAX -> relax_directive
| tokens -> sentence_directive
| CHANGE tokens_identifier_no_to TO tokens atomic_change_slides atomic_change_k
  -> change_directive
| SET tokens_identifier_no_to TO tokens -> set_directive

atomic_change_slides: ["in" tokens "slides"]
atomic_change_k: ["with" "k" "of" tokens]

tokens: (_token|expression)+
tokens_identifier: _token_no_values _token*
tokens_identifier_no_to: _token_no_to_values _token_no_to*
_token: _token_no_to | TO
_token_no_values: _token_no_to_values | TO
_token_no_to_values: IDENTIFIER | VARIABLE | SCOPE_OPERATOR

// Rules for expressions
expression: "(( " expr_ternary _DUB_RPAR
?
expr_ternary: expr_or
| expr_or "?" expr_ternary ":=" expr_or -> expr_cond
?
expr_or: expr_and
| expr_or "||" expr_and
?
expr_and: expr_not
| expr_and "&&" expr_not
expr_not: expr_comp -> expr_inline
| "!" expr_not
?
expr_comp: expr_sum
| expr_sum ">" expr_sum -> expr_gt
| expr_sum "<" expr_sum -> expr_lt
| expr_sum "=" expr_sum -> expr_eq
| expr_sum ">=" expr_sum -> expr_ge
| expr_sum "<=" expr_sum -> expr_le
| expr_sum ">>=" expr_sum -> expr_ne
?
expr_sum: expr_product
| expr_sum "+" expr_product -> expr_add
| expr_sum "-" expr_product -> expr_sub
?
expr_product: expr_atom
| expr_product "*" expr_atom -> expr_mul
| expr_product "/" expr_atom -> expr_div
| expr_product "/" expr_atom -> expr_mod
?
expr_atom: (UNSIGNED_REAL|UNSIGNED_INTEGER|string|BOOLEAN) -> expr_value
| "=" expr_atom -> expr_neg
| tokens_identifier -> expr_var
| 
actor_header: tokens_identifier
actor_header: "actor_header" ("=" number_spec ")? [COLON _actor_def_property_part] actor_footer -> actor_def_with_inheritance
| actor_header ("<" CONSTRUCTOR ">")? COLON _actor_def_property_part actor_footer

actor_def: actor_header ("(" inheritance ")") ("="<" CONSTRUCTOR ">")? [COLON _actor_def_property_part] actor_footer

actor_def_property_part: _NL _INDENT actor_property ([_NL] actor_property)* [_NL] _DEDENT
| actor_property

CONSTRUCTOR: constructor_param ("",") constructor_param*
constructor_param: tokens_identifier ("=" tokens_identifier ")"
constructor_call: (tokens ("",") tokens))?

inheritance: single_inheritance ("",") single_inheritance)*
single_inheritance: tokens_identifier ("=" constructor_call [(COLON | BUT) inheritance_property ("",") inheritance_property)* ">")?
inheritance_property: tokens_identifier initializer

sub_actor_mark: "dummy"-0

has_or_has_meta: HAS | HAS META

actor_property: has_or_has_meta tokens_identifier ("=" single_inheritance ")" [initializer] ->
| has_property
| has_or_has_meta tokens_identifier initializer -> has_property_inference
| CAN tokens_identifier COLON _NL _INDENT can_variables modifiers _directives
| _DEDENT -> can_property
| actor_def -> sub_actor

initializer: "=" tokens
| "=" tokens -> initializer_lazy

modifiers: [modifier ([_NL] modifier)*] [_NL]

modifier: "=" tokens ""
| "=" tokens COLON _NL _INDENT _directives _DEDENT -> modifier_with_directives

can_variables: [_NL] [ "=" [can_variable ("",") can_variable)* "]" _ NL]
can_variable: tokens_identifier "=" tokens

// Terminals for whitespace
_INDENT: "<INDENT>"
_DEDENT: "<DEDENT>"

// Terminals for keywords/punctuation
COLON.5: ":="
MEANWHILE.10: "meanwhile"i
THEN.10: "then"i
CALL.10: "call"i
RELAX.10: "relax"i
CHANGE.10: "change"i
Listing A.2: test_smalltalk.py: Demonstration of context implementation of the Smalltalk type hierarchy

```python
import unittest
from thapl.context import Compound, Named
from thapl.pattern import Pattern
from thapl.utility import one_or_raise

def create_smalltalk():
    objects = {
        "class": Compound(name="class", ), actor=True),
        "object": Compound(name="object", ), actor=True),
        "magnitude": Compound(name="magnitude", ), actor=True),
        "number": Compound(name="number", ), actor=True),
        "integer": Compound(name="integer", ), actor=True),
        "var": Compound(name="var", ), actor=True)
    }

    # set classes for everything
    objects["class"].append_actor(Named(Pattern(["class"]), objects["class"]))
    objects["object"].append_actor(Named(Pattern(["class"]), objects["class"]))
    objects["magnitude"].append_actor(
```
Named(Pattern("class")), objects["class"])
objects["number"].append_actor(Named(Pattern("class")), objects["class"))
objects["integer"].append_actor(Named(Pattern("class")),
objects["class"))
objects["var"].append_actor(Named(Pattern("class")), objects["integer"))

# set inheritance
objects["magnitude"].append_actor_inherited(objects["object"])
objects["number"].append_actor_inherited(objects["magnitude"])
objects["integer"].append_actor_inherited(objects["number"])

# set method for example
objects["object"].append_actor(
    Named(Pattern("get", "class")),
    Compound(command=Lambda self: self.find_actor(["class"])))

RETURN objects

DEF deref(g):
    RETURN one_or_raise(g, LookupError)[0]

CLASS TestLittleSmalltalk(unittest.TestCase):
    DEF test_get_class(self):
        smalltalk = create_smalltalk()
        var_class = deref(smalltalk["var"].findActor(["class"]))
        var_get_class = deref(var_class.findActor(["get", "class"]))
        self.assertIn(smalltalk["integer"],
                          deref(var_get_class.begin())(smalltalk["var"])))
Appendix B

Code Listings for \TeX{} and \texttt{Thapl} Examples

B.1 Towers of Hanoi

B.1.1 \LaTeX{}

This example is taken verbatim from the \TeX{} examples website [Damman and Hofmann, 2010].

Listing B.1: \LaTeX{} Implementation of Towers of Hanoi

\begin{verbatim}
\% Towers of Hanoi illustrated and computed by \TeX{}. The problem is solved in \TeX{} and for every move the situation is drawn. The idea and visualization were \% by Martin Hofmann, Berteun Damman programmed the actual recursion.
\%
\% Authors: Martin Hofmann and Berteun Damman

\% This is needed for a hyperref/beamer interaction bug
\RequirePackage{atbegshi}
\documentclass{beamer}
\usepackage{scalefnt}
\hypersetup{pdfpagemode=FullScreen}
\usetheme{Warsaw}

\usepackage{tikz}
\usetikzlibrary{shadows,patterns,shapes}
\%
\% The logic for Hanoi, we record the discs at every pole
\% as a comma separated list ending with a \'.\'; i.e. the
\%
\% starting list for 4 discs would be 1,2,3,4,.
\newcount\discs
\def\initpoles#1{
  \def\disclist{}\def\endcsname{\disclist.}
  \foreach \n in {1,...,#1} {\xdef\disclist{\disclist,\n,}}
  \expandafter\xdef\csname pole 1\endcsname{\disclist.}
  \expandafter\xdef\csname pole 2\endcsname{.}
\}
\def\p1\disclist\n\def\p2\disclist.\endcsname
\end{verbatim}
\expandafter\def\csname pole 3\endcsname{.}
}

% Delimited macro; #1 is everything up to the first ',' and
% #2 everything after it.
\def\head#1,#2.{#1}
\def\tail#1,#2.{#2}

% This macro updates the disc lists, its arguments are the name
% names of the macro's corresponding to the poles, for example
% 'pole 1' and 'pole 3'.
\def\movedisc#1#2{
    \edef\lista{\csname #1\endcsname}
    \edef\listb{\csname #2\endcsname}
    \expandafter\xdef\csname #2\endcsname{\expandafter\head\lista,\listb}
    \expandafter\xdef\csname #1\endcsname{\expandafter\tail\lista.}
}

% Updates the lists and then draws a new frame.
\def\move#1#2{
    \movedisc{pole #1}{pole #2}
    \gdef\fmsg{\node[anchor=north] at (3.8,-.5) {Moved disc from pole #1 to pole #2.};}
    \drawpoles
}

% This macro boils down to a well-known recursive solution, as given
% here for example:  http://en.wikipedia.org/wiki/Towers_of_Hanoi#Recursive_solution
%
% #1 Pole to move from
% #2 Pole to move to
% #3 Pole to use as scratch
% #4 Number of disks
\def\rhanoi#1#2#3#4{
  \ifnum#4>1
    \advance#4 by -1 \rhanoi#1#3#2#4
    \move{#1}(#3)
    \advance#4 by -1 \rhanoi#2#1#3#4
  \else
    \move{#1}(#3)
  \fi
}

% Below is the TikZ code to draw the towers:
\tikzset{
  disc/.style={shade, shading=radial, rounded rectangle, minimum height=.5cm,
                inner color=#1!20, outer color=#1!60!gray},
  disc 1/.style={disc=yellow, minimum width=15mm},
  disc 2/.style={disc=orange, minimum width=20mm},
  disc 3/.style={disc=red, minimum width=25mm},
  disc 4/.style={disc=green, minimum width=30mm},
  disc 5/.style={disc=blue, minimum width=35mm},
  disc 6/.style={disc=purple, minimum width=40mm},
}

% Define some colors, I don't like plain green and brown.
\definecolor{darkgreen}{rgb}{0.2,0.55,0}
\definecolor{darkbrown}{rgb}{0.375,0.25,0.125}
newcommand\pole{\fill[darkbrown] (-1.6cm, 0) rectangle (1.6cm,0.25cm)
(-1.25mm,2.5mm) rectangle (1.25mm,4.25cm);
}

% Because the list starts with the topmost disc, we
% use two recursive macro’s to invert the drawing process.
\newcounter\curlevel

% This macro checks whether the list is empty, if not,
% it calls \rdrawdiscs which removes one element and
% calls this one again.
\def\drawdiscs#1{\def\n{\THE\curlevel}
% If #1 is empty, this expands to \if. which is true, otherwise
% we're safe to assume there's at least one element.
\expandafter\if\n#1..\else
\rdrawdiscs#1.
% ADVANCE\curlevel by 1\relax
\fi
}

\def\rdrawdiscs#1,#2.{\def\n{\THE\curlevel}
% Draw the actual disk.
\node[disc #1,yshift=\n*5mm] {#1};
}

\def\discs#1{\curlevel=1
\expandafter\drawdiscs#1
}

% Draws the whole situation based on the lists.
\def\drawpoles{\begin{frame}{\ftitle}
\begin{tikzpicture}
\foreach \n/\x in {1/0cm,2/3.8cm,3/7.6cm} {\begin{scope}[xshift=\x]
pole
\expandafter\discs\csname pole \n\endcsname
\end{scope}
}
% Macro that either contains something like OK or
% the last move.
\fmsg
% We use this to prevent the picture from jumping between
% frames.
\useasboundingbox (-1.6cm,-1.2cm) rectangle (9.2cm,4.25cm);
\end{tikzpicture}
\end{frame}
}

% Main macro, initiates the lists for the current number, sets a title
% for the frame and starts the recursion.
\def\hanoi#1{\ndiscs=#1

B.1.2 THAPL

Listing B.2: Complete THAPL implementation of Towers of Hanoi

```
PLAY hanoi
LOAD library\animable, library\location from "library.thapl"

CHARACTERS
rectangle (animable "'(start) rectangle (end)""):
  HAS start (location)
  HAS end (location)
pole (animable "\\begin{scope}[xshift=(xshift)]cm,yshift=(yshift)]cm\end{scope}"") <xs (real)>:
  has rect1 (rectangle <but start\x = -1.6, start\y = 0.0, end\x = 1.6, end\y = 0.25>)
  has rect2 (rectangle <but start\x = -0.125, start\y = 0.25, end\x = 0.125, end\y = 4.25>)
  has brown (flag) = true
  has xshift => xs
  has yshift = -2.0
disc (animable "\\node \[yshift={(yshift)}*5 mm],xshift={(xshift)}cm,\keys\] {\text}) <ys (real), ctext (string), max height (real)>:
  has shade (flag) = true
  has shading = "radial"
  has rounded rectangle (flag) = true
  has minimum height => 0.5
  has meta maximum height => max height
  has xshift = 0.0
  has yshift => ys
```
HAS text => ctext

CAN move:
( time = 9.0, k = 1 )
[slowly]:
  SET time to (( time + 2 ))
[quickly]:
  SET time to (( time / 2 ))
[importantly]:
  SET k to (( k + 10 ))
(to $pole$ at height $yshift$)
CHANGE yshift to (( maximum height )) in (( time / 3 )) slides with k of k
CHANGE xshift to $pole$xshift in (( time / 3 )) slides with k of k
CHANGE yshift to (( $yshift - 4.0 )) in (( time / 3 )) slides with k of k

ACTORS
boundingbox (animable "\useasboundingbox ((rect));":):

HAS rect (rectangle <\begin{tikzpicture} start\x = -2.0, start\y = -0.5, end\x = 10.0, end\y = 5.0>)

poles:
  pole 1 (pole <0.0>)
  pole 2 (pole <3.6>)
  pole 3 (pole <7.2>)

discs:
  disc 1 (disc <-1.0, "1", 8.0>):
    HAS minimum width = 30.0
    HAS inner color = "green!20"
    HAS outer color = "green!60!gray"
  disc 2 (disc <((-2.0 ), "2", (( disc 1\maximum height - 1.4 )))>):
    HAS minimum width = 35.0
    HAS inner color = "blue!20"
    HAS outer color = "blue!60!gray"
  disc 3 (disc <-3.0, "3", (( disc 2\maximum height - 1.4 )))>):
    HAS minimum width = 40.0
    HAS inner color = "purple!20"
    HAS outer color = "purple!60!gray"

ACTION
discs\disc 1 move slowly slowly to poles\pole 2 at height 1 MEANWHILE discs\disc 2 move slowly importantly to poles\pole 3 at height 1
discs\disc 1 move to poles\pole 3 at height 2 MEANWHILE discs\disc 3 move to poles\pole 2 at height 1
discs\disc 1 move to poles\pole 1 at height 1 MEANWHILE discs\disc 2 move to poles\pole 2 at height 2
discs\disc 1 move to poles\pole 2 at height 3

B.2 Simulating a Logic Circuit

B.2.1 $\LaTeX$

This example is taken verbatim from the $\TeX$ – $\LaTeX$ Stack Exchange.

Listing B.3: $\LaTeX$ implementation of a flip-flop digital circuit

\documentclass[beamer]
\usepackage[american]{circuitikz}
\usepackage{xcolor}
\definecolor{HIGH}{HTML}{ff0000}
\foreach \C/\S/\R/\NS/\NR/\Q/\NQ/\STAT/\vC/\vS/\vR/\vNS/\vNR/\vQ/\vNQ in {  
  LOW/LOW/LOW/HIGH/HIGH/NC/NC/No Change/0/0/0/1/1/\overline{Q_0}/,  
  LOW/LOW/HIGH/HIGH/HIGH/NC/NC/No Change/0/0/0/1/1/\overline{Q_0}/,  
  LOW/HIGH/LOW/HIGH/HIGH/NC/NC/No Change/0/1/0/1/1/\overline{Q_0}/,  
  LOW/HIGH/HIGH/HIGH/HIGH/NC/NC/No Change/0/1/1/1/1/\overline{Q_0}/,  
  HIGH/LOW/LOW/HIGH/HIGH/NC/NC/No Change/1/0/0/1/1/\overline{Q_0}/,  
  HIGH/LOW/HIGH/HIGH/HIGH/NC/NC/No Change/1/0/0/1/1/\overline{Q_0}/,  
  HIGH/LOW/LOW/HIGH/HIGH/RESET/SET/Reset/1/0/1/0/1/0/\overline{Q_0}/,  
  HIGH/LOW/HIGH/HIGH/HIGH/RESET/SET/Reset/1/0/1/0/1/0/\overline{Q_0}/,  
  HIGH/LOW/HIGH/HIGH/HIGH/HIGH/HIGH/HIGH/HIGH/LOW/LOW/INVALID/INVALID/Invalid/1/0/0/0/0/Q/Q',  
  HIGH/LOW/HIGH/HIGH/HIGH/HIGH/HIGH/HIGH/HIGH/LOW/LOW/INVALID/INVALID/Invalid/1/0/0/0/0/Q/Q'}{  
  \begin{frame}  
  \begin{tikzpicture}  
  \def\NAND(########1)########2[########3]##{}{\begin{scope}[shift={(########1)}]  
  \% Draw the pins  
  \draw[########3,line width=1pt] (-0.5,0.25) -- +(0,0.25) coordinate (########2 IN1);  
  \% Pin 1 IN  
  \draw[########3,line width=1pt] (-0.5,-0.25) -- +(0,0.25) coordinate (########2 IN2);  
  \% Pin 2 IN  
  \draw[########3,line width=1pt] (0.7,0) -- ++(0.25,0) coordinate (########2 OUT);  
  \% Pin 3 OUT  
  \draw[##5,line width=1pt] (-0.5,0) arc (90:-90:0.5) -- cycle; \% The body of IC  
  \node at (0,0) node[align=center]{\small \sf U-########2}; \% IC LABEL  
  \end{scope}  
  }  
  \node[\Q, draw, anchor=west] at (3.5,-1.5) \{}{\sf STATE: \STAT};  
  \NAND(0,0){1}\[(\S)\]\\[\C]\\[\NS]\\[\NR]\\[\Q]\\[\NQ];  
  \NAND(0,-0.25)\{(\S)\}[\Q]\\[\NS]\\[\NR]\\[\Q]\\[\NQ];  
  \NAND(0,-3)\[(\C)\][\NS]\\[\NR];  
  \NAND(0,-2.75)\{(\Q)\}[\NS]\\[\NR];  
  \% LOGIC C WIRE  
  \draw[color=\C,\line width=1pt\{}(-2,-1.5)  
  \node [anchor=south]{\sf C=\vC};  
  \node\[\textcolor{\NS}\text{\small \textit{\overline{S}=\vNS}}\] at (1 IN2)\{\};  
  \% LOGIC S WIRE  
  \draw[color=\S,\line width=1pt\{}(1 IN1)  
  \node [anchor=south]{\sf S=\vS};  
  \% LOGIC R WIRE  
  \draw[color=\NS,\line width=1pt\{}(1 OUT)  
  \node [anchor=south]{\textcolor{\NS}\text{\textit{\overline{S}=\vNS}}\}--(2 IN1);  
  \% LOGIC NOT-S WIRE  
  \draw[color=\NS,\line width=1pt\{}(1 OUT)  
  \node [anchor=south]{\textcolor{\NS}\text{\textit{\overline{S}=\vNS}}\}--(2 IN1);  
  \end{tikzpicture}  
  \end{frame}  

B.2.2 THAPL

Listing B.4: THAPL implementation of a flip-flop digital circuit

```
PLAY hanoi
LOAD library\animable, library\location from "library.thapl"

CHARACTERS
nand gate (animable <<<\begin{scope}[shift=((\shift))], line width=1pt]
\draw((pin 1 color)) (-0.5,0.25) -- +(-0.25,0) coordinate (%1)\%(\name) IN1);
\draw((pin 2 color)) (-0.5,-0.25) -- +(-0.25,0) coordinate \%(\name) IN2);
\draw((pin 3 color)) (0.7,0) -- +(+0.25,0) coordinate \%(\name) OUT);
\draw((pin 3 color)) (0.6,0) circle (2.5pt);
\draw[fill=blue!10] (-0.5,0) |- ++ (0.5,0.5) arc (90:-90:0.5) -| cycle;
\draw[blue] (0,0) node {small \sf \tiny NAND Gate};
\end{scope}<<<

HAS pin 1 = false
HAS pin 2 = false
HAS pin 3 = (( !(pin 1 && pin 2) ))
HAS pin 1 color = (( pin 1 ? "red" : "blue" ))
HAS pin 2 color = (( pin 2 ? "red" : "blue" ))
HAS pin 3 color = (( pin 3 ? "red" : "blue" ))
HAS name (string)
HAS shift (location)
input wire (animable <<<) <in (boolean), out (reference)>:
HAS input (boolean) => in
HAS color => (( input ? "red" : "blue" ))
HAS output => out
CAN on:
   CHANGE input to true MEANWHILE CHANGE output\ to true
CAN off:
   CHANGE input to false MEANWHILE CHANGE output\ to false
input double wire (animable <<<) <in (boolean), out 1 (reference), out 2 (reference)>:
HAS input => in
HAS color => (( input ? "red" : "blue" ))
HAS output 1 => out 1
HAS output 2 => out 2
```
node on:
CHANGE input to true MEANWHILE CHANGE output 1\ to true MEANWHILE CHANGE output 2\ to false

CAN off:
CHANGE input to false MEANWHILE CHANGE output 1\ to false MEANWHILE CHANGE output 2\ to false

patch wire (animable <"">) \ in (reference), out (reference) >:
HAS input => in
HAS color => (\input ? "red" : "blue")
HAS output (reference) => out
output double wire (animable <"">) \ in (reference), out 2 (reference), out v (reference) >:
HAS input => in
HAS color => (\input ? "red" : "blue")
HAS output 1 => input
HAS output 2 => out 2
HAS output value => out v

ACTORS

state (animable <"\node[\{(top color)\}, draw, anchor=west\} at (3.5,-1.5) {\scriptsize\sf STATE: \{(state text)\});">):
HAS top color = "black"
HAS bottom color = "black"
HAS state text = "No change"
HAS top output = "\bullet"
HAS bottom output = "\bullet"
CAN is reset:
CHANGE state text to "Reset" MEANWHILE CHANGE top color to "yellow" MEANWHILE CHANGE bottom color to "green" MEANWHILE CHANGE top output to "0" MEANWHILE CHANGE bottom output to "1"
CAN is set:
CHANGE state text to "Set" MEANWHILE CHANGE top color to "green" MEANWHILE CHANGE bottom color to "orange" MEANWHILE CHANGE top output to "1" MEANWHILE CHANGE bottom output to "0"
CAN is invalid:
CHANGE state text to "Invalid" MEANWHILE CHANGE top color to "orange" MEANWHILE CHANGE bottom color to "orange" MEANWHILE CHANGE top output to "\bullet" MEANWHILE CHANGE bottom output to "\bullet"

\begin{itemize}
\item \textbf{nand 1} (nand gate <\texttt{but} shift|x = 0, shift|y = 0, name = "1", pin 1 = false, pin 2 = false>)\end{itemize}
\begin{itemize}
\item \textbf{nand 2} (nand gate <\texttt{but} shift|x = 3, shift|y = -0.25, name = "2", pin 1 = true, pin 2 = true, pin 1 color => (\texttt{neg s wire}|color ), pin 2 color => \texttt{state}|top color, pin 3 color => \texttt{state}|bottom color)\end{itemize}
\begin{itemize}
\item \textbf{nand 3} (nand gate <\texttt{but} shift|x = 0, shift|y = -3, name = "3", pin 1 = false, pin 2 = false>)\end{itemize}
\begin{itemize}
\item \textbf{nand 4} (nand gate <\texttt{but} shift|x = 3, shift|y = -2.75, name = "4", pin 1 = true, pin 2 = true, pin 1 color => \texttt{state}|bottom color, pin 2 color => (\texttt{neg r wire}|color ), pin 3 color => \texttt{state}|top color)\end{itemize}

s wire (input wire <false, nand 1\pin 1 \texttt{but} rendering = "")
\begin{verbatim}
\end{verbatim}

r wire (input wire <false, nand 3\pin 2 \texttt{but} rendering = "")
\begin{verbatim}
\end{verbatim}

\textbf{c wire} (input double wire <false, nand 1\pin 2, nand 3\pin 1 \texttt{but} rendering = "")
\begin{verbatim}
\end{verbatim}

\textbf{neg s wire} (patch wire (<\texttt{nand 1}\pin 3 ), (\texttt{nand 2}\pin 1 )) \texttt{but} rendering = "")
\begin{verbatim}
\end{verbatim}

\textbf{neg r wire} (patch wire (<\texttt{nand 3}\pin 3 ), (\texttt{nand 4}\pin 2 )) \texttt{but} rendering = "")
\begin{verbatim}
\end{verbatim}

96
ACTION
s wire on
s wire off MEANWHILE c wire on
r wire on MEANWHILE state is reset
r wire off MEANWHILE s wire on MEANWHILE state is SET
r wire on MEANWHILE state is invalid

B.3 Hannukah Menorah

B.3.1 THAPL

Listing B.5: Complete Hannukah Menorah THAPL implementation

\begin{verbatim}
PLAY hannukah
LOAD library\animable from "library.thapl"
CHARACTERS
  candle (animable ""
\begin{scope}[yshift=(height)mm]
  \draw[line width=.3mm, line join=miter, miter limit=20] (1)(3) in (pos)*6mm,5mm
  to[pos=0, in=-45] ++(0,5mm) to[pos=0, in=180] ++(0,-5mm)
  to[pos=20, in=45] ++(.3mm,2mm) to[out=90, in=160] ++(-3mm,-2mm);
\end{scope}"

HAS height = 0

ACTORS
  body (animable ""
  \foreach \pos in {1,...,4}{
    \fill (\pos + 6mm,2.5mm) ++(1mm,0) -- ++(1mm,2mm) -- ++(-4mm,0) -- ++(1mm,-2mm) -- cycle
    (\pos + 6mm,2.5mm) ++(1mm,0) -- ++(1mm,2mm) -- ++(-4mm,0) -- ++(1mm,-2mm) -- cycle;
  }
  \fill (0,5mm) ++(1mm,0) -- ++(2mm,0) -- ++(2mm,0) -- ++(0,1mm) -- ++(0,1mm) -- cycle;
  \fill (0,5mm) ++(-10mm,-2mm) -- ++(0,-1mm) -- ++(20mm,0) -- ++(0,1mm) -- ++(0,1mm) -- cycle;
  \fill (0,5mm) -- ++(-1mm,0) -- ++(0,1mm) -- ++(1mm,0) -- ++(1mm,0) -- ++(0,1mm) -- ++(0,1mm) -- cycle;
  \end{scope}

shamash (animation ""
  \foreach \pos in {-4,...,-1,1,...,4}{
    \fill (\pos + 5mm,2.5mm) ++(1mm,0) -- ++(1mm,2mm) -- ++(-4mm,0) -- ++(1mm,-2mm) -- cycle
    (\pos + 5mm,2.5mm) ++(1mm,0) -- ++(1mm,2mm) -- ++(-4mm,0) -- ++(1mm,-2mm) -- cycle;
  }
  \fill (0,5mm) -- ++(-1mm,0) -- ++(0,1mm) -- ++(1mm,0) -- ++(1mm,0) -- ++(0,1mm) -- ++(0,1mm) -- cycle;
  \end{scope}
\end{verbatim}
B.4 Convolution

B.4.1 THAPL

Listing B.6: Convolution presentation: THAPL implementation

```
DECLARE
  has pos = 0
ACTION
  action
  
  -4出现 candles disappear
  -3出现 candles disappear
  -2出现 candles disappear
  -1出现 candles disappear
  1出现 candles appear
  2出现 candles appear
  3出现 candles appear
  4出现 candles appear

PLAY convolution

LOAD library\animable, library\measure, library\location from "library.thapl"

CHARACTERS

axis label (animable <"\node[\{\\keys\}] at (2,0) {\$\scriptstyle (text)\$};">):
  HAS text = ""

y label (animable <"\node[\{\\keys\}] at ((loc\x),(loc\y)) { (\label) };">) <1
  (string)>:
  HAS loc (location)
  HAS semithick (flag) = true
  HAS anchor = "east"
  HAS label = 1

axis (animable <"\node[\{main line\}](x tick marks)(\{y tick mark\})(\{y label 1\})(\{y label 2\})">):
  HAS main line (animable <"\foreach \x/\label in {-2/llap{$-2$}2/llap{$2$}}1/0,0,1,2/2} { \node[anchor=north,minimum width=7mm] at (\x,0) {\$\label\$}; \draw[semithick](\x,0) -- (\x,-1.25pt); }

  "">):

y tick mark (animable <"\node[\{\keys\}] at (-1.25pt,.5) -- (0,.5);">):
  HAS y label 1 (y label <"$\frac{1}{2}$" BUT loc\x = 0.0, loc\y = 0.0)
  HAS y label 2 (y label <"$1$" BUT loc\x = 0.0, loc\y = 1.0)
  HAS label (axis label)

scopred axis (animable ""
```

98
\begin{scope}[yshift=((yshift))]
  \draw<+[-0.5,0]>(axis) coordinate (s) <+-0.5,0,1cm> -- (s);
  \draw<+[-0.5,0]>(axis) coordinate (t) <+-0.5,0,1cm> -- (t);
  \draw<+[-0.5,0]>(axis) coordinate (f) <+-0.5,0,1cm> -- (f);
  \draw<+[-0.5,0]>(axis) coordinate (g) <+-0.5,0,1cm> -- (g);
\end{scope}

B.5 Intersection

B.5.1 THAPL

Listing B.7: Intersection presentation: THAPL implementation
\draw[semithick] (\x, 0) -- (\x, -1.25pt);
}

HAS y tick mark (animable \"\draw[semithick] (-1.25pt, .)\" \& \(0, .5\))
HAS y label 1 (y label \"\frac{1}{2}\" BUT loc\(x = 0.0, \loc y = 0.5\))
HAS y label 2 (y label \"1\" BUT loc\(x = 0.0, \loc y = 1.0\))
HAS label (axis label)
 Scoped axis (animable \"\begin{scope}[yshift=\(\ys\)]
\end{scope}\")

HAS axis (axis)
HAS yshift (measure \& length = 0.0, unit = \"cm\")

line (animable \"\draw[\(\_\_\\\_\keys\),\blue!50] (\(\(\text{start \(x\)}\),\(\(\text{start \(y\)}\)\)\cm - 2.0\cm) -- (\(\(\text{\(x\)}\),\(\(\text{expression for \(x\)}\)\)\cm);\") \(< s \(x \text{ (real)}, s \(y \text{ (real)}, x \text{ expr (string)}\):\
HAS thick (flag) = true
HAS x (real)
HAS start x => s x
HAS start y => s y
HAS expression for x => x expr
CAN stretch:
[x to \$target]
CHANGE x to \$target

ACTORS

tau axis (scoped axis \& axis\label\text = \"\tau\")
t axis (scoped axis \& axis\label\text = \"t\", yshift\length = -2.0)
f tau (animable \"\draw[\textcolor{red}{50}] (-0.5, 0) -- +(0,1) -- +(1,1) -- +(1,0);\clip (-2,-2) rectangle (2,1.75);\")
g tau (animable \"\draw[\textcolor{green}{50}] (\(\(\text{\(x\)}\),0) ++(-0.5, 0) -- +(0,1) -- +(1,1) -- +(1,0);\draw[\textcolor{green}{50}\text{dashed}] (\(\(\text{\(x\)}\),1.4) -- (\(\(\text{\(x\)}\),-2.0\cm);\node[fill=white] at (\(\(\(\text{\(x\)}\), 1.25) \{g\};\")

at (\(\(\text{\(x\)}\), 1.25) \{g\};\")

has x (real) = -2.0
CAN step:
CHANGE x to (( x + 1.0 ))

line left (line <\(-1.0, 0.0, \text{"-1cm"}" BUT x = -1.0>)
line right (line <\(-1.0, 1.0, \text{"1cm"}" BUT x = 0.0>)

ACTION

g tau step
g tau step MEANWHILE line left stretch x to 0.0
g tau step MEANWHILE line right stretch x to 1.0
g tau step
Appendix C

Syntax Derivation Rules

\[\text{Program} \Rightarrow \text{Play}^* \text{ Script} \quad (C.1)\]
\[\text{Play} \Rightarrow \text{PLAY Act}^* \text{ Name Script} \quad (C.2)\]
\[\text{Act} \Rightarrow \text{ACT Scene}^* \text{ Name Script} \quad (C.3)\]
\[\text{Scene} \Rightarrow \text{SCENE Name Script} \quad (C.4)\]
\[\text{Script} \Rightarrow \text{Definitions}_{\text{opt}} \text{ Directives}_{\text{opt}} \quad (C.5)\]

\[\text{Definitions} \Rightarrow \text{Characters}_{\text{opt}} \text{ Actors}_{\text{opt}} \quad (C.6)\]
\[\text{Actors} \Rightarrow (\text{DRAMATIS PERSONAE}|\text{ACTORS}) \text{ Actor Specification}^* \quad (C.7)\]
\[\text{Characters} \Rightarrow \text{CHARACTERS Actor Specification}^* \quad (C.8)\]

\[\text{Directives} \Rightarrow \text{ACTION Directive} \quad (C.9)\]

\[\text{Entity Definition} \Rightarrow \text{Name ( Array Spec)}_{\text{opt}} < \text{Constructor} > \text{ Inheritance List}_{\text{opt}} \left( : \text{Property}^+ \right)_{\text{opt}} \quad (C.10)\]
\[\text{Inheritance List} \Rightarrow ( \text{Inheritance Clause} , \text{Inheritance Clause})^* \quad (C.11)\]
\[\text{Constructor} \Rightarrow \text{Constructor Parameter} ( , \text{Constructor Parameter})^* \quad (C.11)\]
\[\text{Constructor Parameter} \Rightarrow \text{Parameter ( Type )} \quad (C.12)\]
\[\text{Array Spec} \Rightarrow [ \text{Array Spec Item} ( , \text{Array Spec Item})^* ] \quad (C.12)\]
\[\text{Array Spec Item} \Rightarrow \text{Number ( .. Number)}_{\text{opt}} \quad (C.12)\]

\[\text{Inheritance Clause} \Rightarrow \text{Supertype Name} < \text{Constructor Call} (( : | \text{BUT} ) \text{ Override})_{\text{opt}} > \quad (C.13)\]
\[\text{Constructor Call} \Rightarrow \text{Constructor Call Item} ( , \text{Constructor Call Item})^* \quad (C.14)\]
\[\text{Override} \Rightarrow \text{Inheritance Key Value} ( , \text{Inheritance Key Value})^* \quad (C.15)\]
\[\text{Constructor Call Item} \Rightarrow \text{Value} | \text{Inheritance Key Value} \quad (C.15)\]
\[\text{Inheritance Key Value} \Rightarrow \text{Key} = \text{Value} \quad (C.15)\]
\[ \text{Property} \Rightarrow \text{Has Property} \mid \text{Can Property} \mid \text{Nested Entity} \quad \text{(C.16)} \]

\[ \text{Has Property} \Rightarrow \text{HAS meta} \_\text{opt} \text{ Property Name ( Inheritance Clause )} \]
\[ \mid \text{HAS meta} \_\text{opt} \text{ Property Name ( ( Inheritance Clause ) } \text{opt} \mid = \mid =\Rightarrow \text{ ) Value} \quad \text{(C.17)} \]

\[ \text{Can Property} \Rightarrow \text{CAN Property Name : Can Variables} \_\text{opt} \text{ Modifier} \_\text{opt} \text{ Directive} \quad \text{(C.18)} \]
\[ \text{Can Variables} \Rightarrow \{ \text{ Can Variable} \mid \text{ Can Variable} \}_\text{opt} \quad \text{(C.19)} \]
\[ \text{Can Variable} \Rightarrow \text{ Variable Name ( Type Name )} \quad \text{(C.20)} \]

\[ \text{Nested Entity} \Rightarrow \text{Entity Definition} \quad \text{(C.21)} \]

\[ \text{Directive} \Rightarrow \text{Atomic Directive} \mid \text{Compound Directive}, \quad \text{(C.22)} \]

\[ \text{Compound Directive} \Rightarrow \text{Sequential Directive} \mid \text{Parallel Directive} \quad \text{(C.23)} \]
\[ \text{Parallel Directive} \Rightarrow ( \text{Directive MEANWHILE} )^+ \quad \text{(C.24)} \]
\[ \text{Atomic Directive} \Rightarrow \text{Sentence} \quad \text{(C.26)} \]
\[ \text{Sentence} \Rightarrow ( \text{Subject }, )^+ \text{ Modifier} ^* \text{ Verb Modifier} ^* \quad \text{(C.27)} \]

\[ \text{Change Directive} \Rightarrow \text{CHANGE Subject to Literal or Reference} \]
\[ \text{( in Literal or Reference slides\opt \quad (with k of Literal or Reference\opt \quad \text{(C.28)} } \]

\[ \text{Set Directive} \Rightarrow \text{SET Subject to Literal or Reference} \quad \text{(C.29)} \]

\[ \text{Relax Directive} \Rightarrow \text{RELAX} \quad \text{(C.30)} \]

102
Relax Directive $\Rightarrow$ CALL Name

(C.31)
Appendix D

Installation Guide & Quick Start

This appendix contains instructions on how to obtain and install Thapl, as well as run the provided examples. The instructions in this appendix are summarized in the installation script shown in Listing D.2.

Requirements

5. Computer running Linux. This software was mainly tested on Linux. It is expected that it will also work well on Windows and Mac OS X with minimal if any changes, but this has not been as thoroughly tested.

6. Python 3 (tested on versions 3.7.5 and 3.8.1) with development header files installed (python3-dev or similar.)

7. Jupyter, if using the Jupyter extension.

8. A working LaTeX installation (tested on TeX Live 2019.) This is only required for compiling the examples to PDF or using the Jupyter extension, Thapl does not use LaTeX directly.

D.1 Installing and running the main program

The simplest way to install Thapl is from the GitHub repository using pip.

First, we create a Python venv (virtual environment). This will create a separate directory in which Thapl and its required packages will be installed, so that we will not mess up our system Python environment.

$ python3 -m venv thapl_venv

If when creating the venv a message is displayed about needing to install a system package (e.g., python3-venv), follow the instructions in the message and try the command again.

Next, we enter the venv:

$ source thapl_venv/bin/activate
Though not always required, it is a good idea to upgrade PYTHON’s system packages. This can avoid certain installation error messages.

$ python -m pip install --upgrade pip
$ python -m pip install --upgrade setuptools wheel

Now we can install THAPL through git:

$ python -m pip install git+https://github.com/chaosite/thapl.git

And that’s it! THAPL is now installed and can be used. For example, let’s try running the Hannukiah example. We’ll have to get it from git first.

$ git clone https://github.com/chaosite/thapl.git
$ python -m thapl.main thapl/Examples/hannukah.thapl

We should now see the \LaTeX{} output of the Hannukiah animation. We can also generate the PDF using the makefile:

$ cd thapl/LaTeX
$ make hannukah.pdf

This command will generate the PDF for this example and attempt to display it using evince. To generate the PDF files of all of the provided examples, we can run make with no arguments:

$ cd thapl/LaTeX
$ make

Since this operation compiles all of the examples in sequence, it may take a while.

## D.2 Installing the Jupyter extension

If JUPYTER is not already installed, it can install it using the system package manager (e.g., apt or brew), or using pip:

$ python -m pip install jupyter

Installing the JUPYTER extension is also accomplished using pip. Make sure you are inside the previously created venv, then execute the following command:

$ python -m pip install git+https://github.com/chaosite/ipython-thapl.git

This will have installed the JUPYTER extension. JUPYTER can now be started using the jupyter-notebook command. The extension is loaded using the “magic” JUPYTER command %load_ext thaplmagic. After running this command in the first cell, it is possible to execute THAPL code using %%thapl. For example, try running the code in Listing D.1.
Listing D.1: Example code to run within JUPYTER

```thapl
PLAY

definite pi

acción <r (string)>


\counter = 0 

%%thapl --format svg -x \newcounter{c} -p calc,ifthen

\begin{verbatim}

\begin{figure}[h]
\centering
\begin{tikzpicture}
\newdimen\radius
\radius=3cm
\newcount\counter
\counter=0
\newcounter{c}
\newcommand{\initial}{0}
\newcommand{\final}{0}
\foreach \i in {0,1,...,\counter}
\do
\pgfmathtruncatemacro{\thec}{\i+1}
\draw[red,thick] (\initial) -- (\final);
\draw[cyan,thin] (0,0) -- (\initial);
\stepcounter{c}
\node[blue,above] at (0:3.125cm) {$n = \i$};
\end{verbatim}
\caption{Example figure}
\end{figure}

%%thapl
```

Listing D.2: Installation script for THAPL

```
# This script will install thapl for you
command -v git || sudo apt-get install git python3 python3-dev
command -v python3 || sudo apt-get install python3 python3-dev
python3 -m venv thapl_venv
source thapl_venv/bin/activate
python -m pip install --upgrade pip
python -m pip install --upgrade setuptools wheel
python -m pip install git+https://github.com/chaosite/thapl.git
git clone https://github.com/chaosite/thapl.git
python -m thapl.main thapl/Examples/hannukah.thapl
(cd thapl/LaTeX; make hannukah.pdf)
(cd thapl/LaTeX; make)
python -m pip install jupyter
python -m pip install git+https://github.com/chaosite/ipython-thapl.git
```

# This script will install thapl for you
command -v git || sudo apt-get install git python3 python3-dev
command -v python3 || sudo apt-get install python3 python3-dev
python3 -m venv thapl_venv
source thapl_venv/bin/activate
python -m pip install --upgrade pip
python -m pip install --upgrade setuptools wheel
python -m pip install git+https://github.com/chaosite/thapl.git
```
In the course of the research, several achievements were obtained. The first among them is the preparation of a thesis, which is the basis of the research. The thesis includes detailed documentation of the language, a draft for new programmers, a survey of the literature, a list of the language's characteristics, the development of a mathematical model and examples of code. In addition, the thesis presents the results of the research.

It is possible to use the documentation to create code examples given in the thesis, and in this manner it is possible to write additional presentations. The documentation contains code in the Python language, and it is possible to create presentations using the documentation and also using the Jupyter notebook environment.

Finally, in order to validate the work done, an attempt was made to check the quality of the language's correspondence with the problem-solving solutions on the Stack Exchange site, problems encountered by users of the site. In addition, research was conducted among several programmers, and LaTeX and a subset of questions in the Thapl language were determined.

The notes of the thesis were written and checked in the Thapl-LA-TEX and LA-TEX environments, using the documentation to write and edit text, and in addition, the documentation was used to create Presentation macros and other elements.
The most important characteristics that distinguish this language compared to other programming languages are:

1. **Syntax**: The syntax is flexible, designed to resemble natural language, specifically, the English language. This means it is necessary to refer to syntax in writing reports in the language, but when reading, it becomes mostly code reading, and the reader can easily understand it in English, which makes it easier to understand the reports written in the language. For example, the Hebrew version of the sentence was written as: "The chick moves two steps left, and at the same time, the color of the chick changes to green." In English, this sentence would look like: "chick moves two steps left MEANWHILE chick changes color to green.

2. **Automatic Reflection**: This feature allows the reader to see how the code is written in the language, with the help of a metaphor that is also mentioned in the code.

3. **Choice of Nouns**: This feature allows for the possibility to write the actions of the sequence, with the help of simple sentences in the English language, allowing for the possibility to write the actions in a declarative manner, with the help of simple sentences in the English language, allowing for the possibility to write the actions in a declarative manner, with the help of simple sentences in the English language.

4. **Reflection**: The feature allows for the ability to implement changes and transformations in the document. The meaning is that when changes are made, the reader has to adapt to the changes. In other words, the reader has to change the document in the same way, unless it is specified that he/she remains static, as is the case with the Thapl acronym.

5. **Manipulation of Data Structures**: The usage of Thapl is another important feature in this language. The meaning is that a data structure is defined in the language, with the help of a metaphor that is also mentioned in the code.
The main goal of the research is to delve deeply into the paradigms of declarative and imperative languages of programming. In this regard, we developed a type of programming language focused on the field, named Thapl, intended to create animations in presentations for educational purposes. The language is built around a metaphor of a script in a theater, such as William Shakespeare’s plays. This metaphor guides the overall structure of the language, expressed in a well-organized manner in sections, where sections define the actions and what the objects are assigned to do on the screen. From there, the presentation is divided into scenes, images, and so on. This structure helps to read text in the language and makes it easier for the creator of the presentation to change the animation in the future.

There exist tools available for creating presentations in the field of animation design. All of them are not intended to create animations, and their support is very limited, and it is focused on the entities for animation only. An example of this is the disappearance of the objects, which causes the text to appear in parts. The opposite makes it possible to create animations that exist on their own, such as an example of a journey on a tree, or a representation of a verb at the beginning of the protocol, which is a fragmented visualization of a single word. From a technical point of view, the language that includes all of these elements, namely the animation PDF, is designed to produce applications in programming languages that belong to the declarative paradigm, in which a statement such as "the characters exist" is independent of each other. In contrast, in the imperative paradigm, the manipulations of the shared state serve as the basis for performing manipulations with the characters, as the subsequent action of the characters is determined by the command they are assigned to execute. As a result, there is no need to store the frames and only relate to the moment when a frame was actually executed.

The programming language was designed to meet several needs, and it was assembled in several parts. First of all, the language allows creating presentations of readings, which are understandable even without a computer. Second, the language needs to be able to switch between presentations. The intention is to be able to save a presentation and change it over a long period, such as several semesters, and only make small changes, such as emphasizing certain points for a specific audience, or to solve errors that were revealed in a previous presentation. Finally, the language needs to allow the use of standard tools in software engineering, such as version control. In other words, the language must be textually.

In conclusion, the language Thapl is a unique tool for creating animations that can be understood even without a computer, while also allowing detailed changes over a long period, and solving errors that may occur. It is a versatile tool that can be used both in education and in software engineering.

Thapl, Technion - Computer Science Department - M.Sc. Thesis 2020-12 - 2020
העבודה בוצעה בהנחייתם המשותפת של פרופסורים יוסי גייל ודוד לורנץ, בכונסולה למידוע

המחiative.

חלק מהתוצאות הброורו כפורסומים כמאמרים או הרצאות ממאמריםhteביתו של מחקר בכונסולה

ובכתבי-

הוסף המאמר המביא של מחבר, אשר גרסאותם העדכניות יותר נמצאות:


תודה

לitori דפנה ומיכאל, אניęk זו תודעה על אוניברסיטת החת窕ות ומחקרים לארץ גודז

בכל מקום הם היו קיצורים

أتي מ坞ים ירבי, פורפ' יוסי גייל ופורפ' דוד לורנץ, על הת▷ודד והכחיחות של מחקר, על

שезультטים את המחקר המוחה.

תודה לאימקים שהנ.Getter להם ופקרה, סזריך שדואל ושאותה להופת

עומס ש 관한,רסו שלם אידיעו על מחקוף המחקר, וה_Mouse הוחתמה על העזרה והיוו

לשם את המסקנה.

ול RECEIVER, תודה לאריה על ההבטחה וה섯יעה של מחקר, ועל זה סשה בת אומי חזרות להתח

ולש ישים לומדים על אוניברסילת מחקר.

הכרת תודעה המסה על התחבורה בין קופר זה. המחקר נסמך בחלקים בא мягкוחות המגעים

של האקדמיה האישית והשתתפים להודעה.

1440.

הכרת תודעה המסה על התחבורה בין קופר זה. המחקר נסמך בחלקים בא мягкוחות המגעים
什פת תכונת תיאטרלית
—
Thapl

היבר על מחקר
לשם מילוי תקף של הדרישות להבנת התואר
 Blasio במדעי המחשב

מניחו

מֶנְי. פָלָד

הובט החופלו של מכון טכנולוגיה לישראל
חרושת תש"פ חיפה
נובמבר 2019
Thapl

מתן י. פלד