Spring 2020

Automated Exercise Generation in Mobile Language Learning

Rayo Verweij
Bard College

Automated Exercise Generation in Mobile Language Learning

By

Rayo Verweij

A Senior Project submitted to
the Divison of Science, Math, and Computing
of Bard College

Annandale-on-Hudson, New York
May 2020

ABSTRACT

The *Language Lion* is an Android application that teaches basic Dutch to English speakers. While mobile language learning has increased exponentially in popularity, course creation is still labor-intensive. By contrast, the *Language Lion* uses a map of Dutch to English lexemes, a context-free grammar, and a modified version of the SimpleNLG sentence realiser to automatically generate semi-random translation exercises for the student. Each component is evaluated individually to find and analyze the particular roadblocks in automated exercise generation for mobile language learning.

Keywords: natural language generation, exercise generation, mobile language learning, student modeling, NT2, Android, Kotlin

ACKNOWLEDGMENTS

First and foremost, I want to thank Sven Anderson, my academic adviser and the supervisor of this project. Whether in his office or over Zoom, his critical insights, constant interest, and ever-present cheerfulness have been invaluable to this project. In addition, his taste in tea is impeccable.

I also want to extend my gratefulness to Peter Filkins and Keith O'Hara for their time and useful insights while on my board, and to Justin Hulbert and the other members of the Mind, Brain, and Behavior Seminar for their thoughts, questions, and ideas about this project.

Big shout-outs to the Bard Debate Union and the Office for International Student and Scholar Services for providing me with some much-needed distractions over the past two months. Your support has been relentless. My deepest gratitude to the College itself, as well, for its warm welcome when I came here and its fierce humanity in times of crisis.

Lastly, of course, my love to all of my friends and family across the pond. Mechteld, Maurits, Juan, Lucila, and Vinn, I sent some hugs—fingers crossed they don't get lost in the mail too.

Table of Contents

List of Figures

List of Tables

List of Listings

DEDICATION

To all my teachers—no algorithm could ever replace you.

Aan al mijn docenten—geen algoritme zal jullie ooit vervangen.

Chapter 1

Introduction

The aim of this Senior Project is to build an Android application that allows a speaker of English to learn basic Dutch, through automatically generated translation exercises.

Over the past decade, the rise of smartphones has created the field of Mobile Language Learning (MLL). Applications like *Duolingo*, *Rosetta Stone*, *Memrise*, and *Babbel* have millions of users trying to learn the basics of a new language. While their applications are highly regarded, they are also very labor-intensive to create, as most courses are built completely manually, either by hired experts or community volunteers. As such, most applications also rely on aggressive advertising or expensive subscription fees to maintain themselves.

Meanwhile, research into natural language processing is thriving. Voice assistants are coupled with all major operating systems, screen readers have become very sophisticated, and a world without machine translation seems unthinkable. While many are understandably apprehensive about handing over control of one's education to algorithms as well, it is a logical next step for mobile language learning to explore the merits of automation. Education has always been a balancing act to provide the most personalization with the least resources, and automated exercise generation allows for the unique possibility to tailor lessons to the individual, rather than forcing many through a generalized curriculum.

This project offers a unique approach to language learning by generating sentences in two

languages at the same time, by combining a novel method of encoding semantic concepts with a custom context-free grammar. This language generation library is used in an application that teaches Dutch, up to the level of basic ordering in a restaurant. Each component of the app is thoroughly evaluated, to make specific recommendations for the future of this field of research.

Chapter 2

Background

There are many different moving parts to designing and building a language learning app. This chapter begins with a survey of modern language learning applications and a general overview of other resources that teach Dutch, after which it will dive into different approaches to generating language learning exercises and tracking student progress.

2.1 Mobile language learning

Research into computer-assisted language learning originated during the Cold War, when American intelligence agencies started to look into efficient ways to teach their agents Russian (Beatty, 2010). While an area of consistent academic interest, its use amongst the general population only exploded with the rise of smartphones and big data, during the 2010s, as the mobile form factor allowed for easy access to casual language learning. Nowadays, searching for Mobile Language Learning (MLL) applications will return dozens of results, with many different approaches that are worth taking a look at.

Claiming over 300 million active users, *Duolingo* is one of the most popular MLL apps (Duolingo, 2020). *Duolingo* offers a wide variety of languages, including Dutch, and divides its courses into themed sections such as "Food" or "Animals" that are themselves divided in 1-8 lessons. When completing all the lessons in a section, the next sections are unlocked, until the student completes

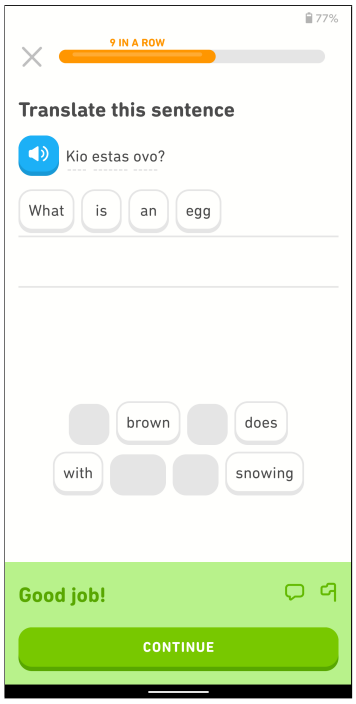

Figure 2.1 An exercise in *Duolingo*.

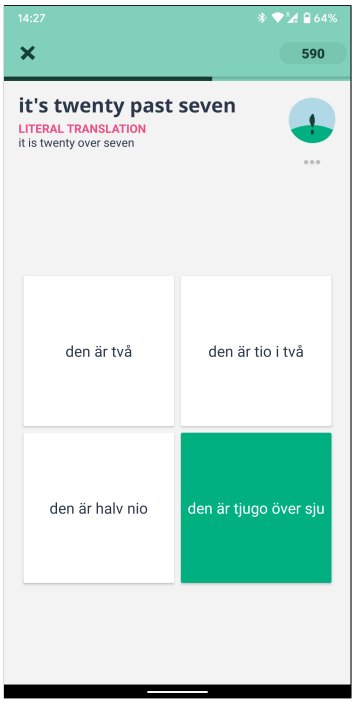

Figure 2.2 An exercise in *Memrise*.

the course. Each lesson is built around three types of exercises: translating sentences directly, building sentences from individual words, and dictation (see Figure 2.1). A few of the most popular language courses have extra features, such as longer stories to read and answer questions about, and podcasts aimed at language learners. These features are not available for Dutch, however. All lessons in *Duolingo* are completely free to use, but a paid subscription plan gives access to features such as offline access.

Except for the most popular courses, which are curated by *Duolingo* employees, all content on the app is written by volunteers. Volunteers build courses by creating the themed sections, writing sentences, and adding all their translations. This community-based approach has allowed *Duolingo* to quickly scale to support many languages and users, although some languages have struggled to attract and retain enough volunteers, such as courses for Finnish and Yiddish, which have been in development for several years.

The MLL application *Babbel* takes a different approach. *Babbel* is a paid service that promises courses built by experts in return (Lesson Nine GmbH, 2020). Compared to *Duolingo*, it offers a

much more tailored approach to language learning, including exercises that try to simulate real-life conversations and detailed explanations of pronunciation. In addition to the linear course, the application includes a personalized "Review" section, which analyzes the student's progress and offers a new list of previously learned words to review each day. *Rosetta Stone* takes this approach even further, in that it is relatively expensive, but provides a comprehensive set of content (promising over 250 hours of curated Dutch lessons alone) and premium features such as a voice recognition system that distinguishes between masculine, feminine, and child voices for better accuracy (Rosetta Stone Ltd., 2020).

On the other end of the spectrum is an app like *Memrise*, which is almost completely based on community content and focuses on learning words and short phrases (Memrise, 2020). Each word or phrase goes through a series of seven exercises, becoming progressively harder, each one visualized as a new stage in planting a seed that grows into a flower (Figure 2.2). While *Memrise* therefore offers an impressive set of vocabulary, these words are always single entities, and never combined into bigger exercises. More curated features, such as listening exercises and videos by locals, can be accessed through a paid subscription.

Next to these relatively well-known applications, a myriad of other options exists, each with their own differentiating feature. *Busuu* relies on its learners to check each other's work, by having them check the assignments of people that are learning their native language in-between their own exercises (Busuu Ltd, 2020). *Drops* is a vocabulary training application that eschews typing completely in favor of tapping or swiping the correct answers, and *Lingvist* has a Course Creator mode which allows the student to enter what words they want to learn, and then picks relevant exercises from its database (Drops, 2020; Lingvist, 2020).

As most of these rely on a paid subscription service to sustain themselves, there are very few open-source MLL applications available. One relatively polished open-source alternative is *10,000 sentences*, which has encoded 10,000 sentences in a number of languages and allows the student to work through them (Krajina, 2020). Other open-source applications often take the form of more traditional flashcard applications, which allow users to build a deck of cards that others can then

use to practice. Compared to their closed-source counterparts, however, these applications are usually more of a vocabulary training companion rather than a full-fledged language course.

While the popularity of the field cannot be denied, it is hard to establish the actual efficacy of these applications, especially as compared to more traditional language learning settings such as classroom education. *Rosetta Stone*, *Duolingo*, and *Babbel* have all commissioned researchers to measure the effectiveness of their applications (Vesselinov, 2009; Vesselinov and Grego, 2012; Vesselinov and Grego, 2016, respectively). Each of these studies were of the same form. A sample of beginner-level learners was instructed to use the Spanish course of the app in question for a set period of time; in the case of *Rosetta Stone*, they were limited to 55 hours, for the other two apps, they were instructed to use the application "often" for two weeks. Before and after the study they took WebCAPE, a placement test commonly used in college language courses, to measure their performance. In each of the cases, the researchers reported an increase in test scores roughly equivalent to one semester of an introductory college-level language class, for *Rosetta Stone* after 55 hours, for *Duolingo* after 34 hours, and for *Babbel* after 21 hours of using the application. Both *Duolingo* and *Babbel* prominently feature these numbers on their web pages and in their marketing.

While these studies seem to show success, they are also directly commissioned by the companies that built the courses and were only focused on one single language. More independent research is few and far between, and often quickly outdated, as was for example a study by Chinnery, which surveyed the possibilities of iPods and PDAs (2004). Steel (2012) did not instruct any group of participants to use a particular MLL application, but rather surveyed close to 600 foreign language students in general. 56% of her respondents reported using a language learning app to complement their university courses, and students reported the ability to practice "anywhere and anytime" as the main benefit of using these applications. "Many" students reported using a variety of language-related applications at the same time, including dictionaries, translators, games, conjugation trainers, and flashcard apps. Steel hypothesizes that MLL is mainly beneficial as a complement to classroom learning, rather than replacing it entirely.

Stockwell and Hubbard (2013) surveyed the entire field of Mobile-Assisted Language Learning

(MALL; "assisted" indicating that includes not only language courses, but also companion apps such as dictionaries and flashcard apps), to identify common pitfalls and formulate a comprehensive set of principles that serve as guidelines for MALL apps to adhere to. Their main principle is to "both distinguish the affordances and limitations of the mobile device and [those] of the environment in which the device will be used in light of the learning target". Their other principles illustrate what is meant by this. Multi-tasking and environmental distractions should be kept to a minimum, to allow the student to concentrate on one task at a time. Push notifications, to remind users of their daily practice, are beneficial, but should respect boundaries[1]. Developers should strive to maintain equity and design their applications to still be usable if the user has limited visual acuity, manual dexterity, internet connectivity, or processing power. The goal of the tasks should be easy to understand, and users should be provided with proper guidance and training when using the app. Tasks should be kept as short and succinct as possible, and the application should properly accommodate other stakeholders in the language learning process as well. For example, when used in a classroom setting, *Duolingo* provides tools for teachers to check their students' progress.

Each of these recommendations was based on a thorough review of the literature, and together they form one of the pillars of this project's evaluation.

2.2 Teaching Dutch as a second language

Before exploring the more technical aspects of automated exercise generation, it is valuable to take a broader look at how Dutch is taught outside of MLL.

There is a small choice of Dutch textbooks available. The default book for immigrants, *Naar Nederland* ("to the Netherlands"), is developed by the Dutch government and available free of

[1] *Duolingo*, in particular, is known for using its expressive owl-mascot *Duo* to pressure its users into resuming practice, by for example showing a cartoon of the owl crying if the user skips a day. On social media, users tend to joke that they are scared of the owl, often reacting to *Duolingo's* posts with comments such as "please let my family go, I promise I will return to practice" and similar.

charge from their website (Ministerie van Sociale Zaken en Werkgelegenheid, 2015). It is designed to prepare any immigrants for the Basic Civil Integration Examination, which is a required test for anyone who wants to become a Dutch citizen, and includes versions for almost 40 source languages, including two 'cleaned' versions that omit stories on for example homosexuality and topless sunbathing, for countries in which possession of books describing these themes is forbidden.

Commercial textbooks are available for a variety of more specific audiences. *De Opmaat* is targeted at college students and is used at universities, *De Delftse Methode* is aimed at learners that have never finished their secondary education, and *DigLin+* is tailored to people that don't know the Latin alphabet (Boom Uitgevers, 2020). While differing in audience and teaching strategies, these books are all maintained by the same publisher, and follow a similar structure of basing their content around a textbook with additional exercises available on a companion website.

As Dutch is a relatively small language, academic research on teaching Dutch as a second language (*NT2* in Dutch) is rare. In general, Dutch is considered to be a hard language to learn. In the first large-scale analysis of NT2 learners, Schepens (2015) analyzed the results of 50,000 participants of the Basic Civil Integration Examination between 1995 and 2010, to investigate the role of linguistic distance between Dutch and the participants' native and second languages in their test performances. While his main conclusions might seem intuitive—speakers of other Germanic languages score higher on Dutch tests than speakers of unrelated languages, such as Chinese or Arabic—his research surfaced a lot of additional information. Kuikens (2017) uses this data to look into the notion that Dutch grammar is very difficult, and while he notes that some aspects of Dutch grammar are relatively simple compared to other languages, such as the absence of tonality and the erosion of cases, there are many features that NT2 learners struggle with regardless of their background. Specifically, he singles out four culprits. Definite articles in Dutch are either "de" or "het" based on the grammatical gender of the following noun, but as nothing in Dutch morphology signals a noun's gender, this must be learned by heart. Compound nouns in Dutch, like German, are not separated by spaces, and can get very complex (look at for example *aansprakelijkheidsverzekeringsuitkering*, or "liability insurance payment"). Dutch word

order is seen as confusing as well, as clause structure will change from subject-verb-object in main clauses to subject-object-verb in subordinate clauses, and the placement of verbs in the perfect tense is fluid. Finally, there is *er*, the Dutch equivalent of "there", which is inserted in many idiomatic expressions, often without a clear clause.

Blom, one of the original authors of *De Delftse Methode*, analyzed his students' performance when given different types of grammatical explanations, and concludes that Dutch grammar is often explained in overly complicated ways, too (1997). While Dutch grammar is full of exceptions and morphophonological rules, he argues that instead of trying to teach the linguistic underpinnings of those rules, students are better served not being taught any explicit rules at all, and instead inferring them themselves from examples, making this the pedagogical ideology behind his textbook.

2.3 Natural language generation

Natural Language Generation (NLG) is the study of artificially constructing human language. Traditionally, NLG has been divided into three distinct parts: content determination, sentence planning, and realization (Reiter & Dale, 1997). Content determination is the process of determining what should be communicated to the user and laying out the rhetorical structure of the document in which the information will be presented. After this, the sentence planning stage divides the content into individual sentences, and decides how those should be linked together, using transition phrases and pronouns. Lastly, the realization phase generates each sentence using correct syntax and interpunction.

In the context of a language learning application, the task of content determination is fulfilled by the pedagogical module, detailed in the next section, and when generating individual exercises, it is not necessary to link sentences together. The main challenge in automated language learning is exercise realization, which needs to happen not just in one language, but in two languages simultaneously, to generate both the question and the answer of an exercise.

While there is an abundance of research into NLG realization for English, the choice for Dutch is severely limited. Academic research is often outdated (see Degand, 1993; Marsi, 1998) or focused on a very specific problem, such as generating reports of football matches (Van der Lee, Krahmer, & Wubben, 2017). The only modern, open-source realization engine that was found to support both English and Dutch is SimpleNLG-NL (De Jong & Theune, 2018).

SimpleNLG-NL is a fork of SimpleNLG, which only supports English (Gatt & Reiter, 2009). The library is written in Java, with ports available for C#, and multiple forks adding support for multiple languages. It allows the user to specify a language and construct sentences in that language by simply setting the different parts of speech. For example, when creating a clause `p`, calling `p.setSubject("he")` and `p.setVerb("walk")` will output "He walks."—complete with capitalization and interpunction.

Sentences can be complicated by setting a variety of features. SimpleNLG-NL automates tasks including inflection of verbs based on tense, voice, and their subject; placement of adjectives and adverbs; handling of Dutch separable complex verbs (cp. "to run away" → "I run away" with "wegrennen" → "ik ren weg"); placement of any complements; building conjunctions; building prepositional phrases; and adding modifiers such as "by the way" to the front of the sentence. As SimpleNLG-NL is itself based on SimpleNLG-EnFr, a fork adding support for French, support for the French language is available as well.

SimpleNLG-NL currently also has some real limitations. First of all, testing showed that word order is often incorrect if a passive, interrogative, or subordinate clause has many complements. Secondly, whereas word order in Dutch is quite flexible, SimpleNLG-NL only ever generates one sentence, even if more possibilities would be correct. Finally, the realizers for the different languages are completely unrelated to one another and generating sentences in two languages simultaneously is therefore not supported.

From here, two approaches can be taken. Either SimpleNLG-NL can be expanded with a module that allows for simultaneous generation, or exercises are generated in one language and a machine translation algorithm finds all different possible answers. A translation system could

go one of two paths: a corpus-statistical approach, which uses a large corpus of data to build probabilistic models that predict a sentence's translation, or a strictly rule-based approach, which defines rules for each exercise that can be given and how to translate it (Russell & Norvig, 2010). While the corpus-statistical approach is not applicable to this situation, as answers need to be generated with absolute certainty, a rule-based approach could technically work. However, it would require a quadruple set of rules: one to generate English, one to generate Dutch, one to generate English from Dutch, and one to generate Dutch from English. A purely NLG-based system seems to be more elegant, and was pursued instead.

2.4 Automated exercise generation

While there have been no studies yet that use natural language generation to generate translation exercises, a few recent projects have come close.

Possibly the most similar research has been by Gilbert and Keet (2018), who aimed to use a NLG-based exercise generator for isiZulu. As isiZulu is a highly complex but severely under-resourced language, this was one of the first efforts to build a sentence realizer for it. Their system included support for a number of exercise types: building sentences from individual words (akin to *Duolingo* in Figure 2.1), pluralizing and singularizing nouns, and conjugating verbs. Each exercise was based around a randomly generated sentence of the pattern `<noun><verb>` or `<noun><verb><noun>`, though the researchers did not achieve full accuracy for the three-word sentences. Each exercise was completely in isiZulu, and translation to or from other languages was not involved.

Burstein and Marcu (2005) built an application to help teachers generate translation exercises from Arabic to English using machine translation. Their application allowed the teacher to specify a specific grammatical construction they wanted exercises for, after which the system would scour two large corpora of Arabic sentences for ones that matched the specification. A machine translation algorithm then automatically generated translations for each sentence, and it was up to the teacher

to select which ones to use and to manually correct the translation, if needed. The researchers reported that their tool allowed teachers to generate exercises 2.6 times faster than without it.

Malafeev (2014) built a generator for open cloze exercises, which are fill-in-the-blank exercises where the learner does not have multiple options, but rather has to come up with the missing word by themselves. These exercises are popular in advanced language learning tests to examine a student's knowledge of idioms and collocations. While the algorithm was concluded to be "quite effective", many of these exercises needed curation by hand as well.

2.5 Student modeling

Generating exercises is only one half of the pedagogical process, however, and the other half is tracking and acting on the student's strengths, weaknesses, and general progress.

In memory research, it is a well-established fact that learning is more effective when the material is divided up in smaller chunks over the course of several days, instead of cramming it all in one long session (Baddeley & Longman, 1978). This type of learning is called *spaced repetition practice* and the challenge remains to find out exactly when each item should be reviewed.

Lamenting that teachers left the study of vocabulary too much in the hands of the students, Pimsleur (1967) jumpstarted research into spaced repetition practice by demonstrating that each time the memory of a word is reconsolidated, it generally remains in memory for exponentially longer than the time before. In the decades since, many attempts have been made at taking this general schedule and making it more personal, taking into account both a learner's profile and the words' individual features. For example, Atkinson (1972) compared four optimization strategies, and settled on a Markov model to predict what word a learner should review next, and Bloom and Shuell (1981) were the first to describe this effect in a classroom setting rather than a fully-controlled study. A meta-analysis of 184 articles on spaced repetition practice by Cepeda et al. (2006) showed decisively that spaced repetition shows benefits even regardless of the specific time interval between reconsolidations, and that there seems to be a ceiling to the length of the interval,

after which further increases have no more positive effect for the learner.

A modern system, named *half-life regression*, has been published by Duolingo (Settles & Meeder, 2016). For each item, this algorithm calculates the probability that it will be correctly recalled as a function of the item's *lag time* and its *half-life*. The lag time is the time since the item was last practiced, and the half-life is an approximation of the item's strength in the learner's memory. It is assumed that the half-life increases exponentially with every review of the item, weighed by the perceived complexity of an item. In other words, a difficult word that has not been practiced much yet will have a much shorter half-life than an easy greeting that has been reviewed a lot. When the lag time is equal to the half-life, the algorithm flags an item as ready for review.

Other research into student modeling has focused on intelligent error-detection systems that give personalized feedback, usually grouped under the name of *intelligent language tutoring systems*, or ILTS. An early example by Heift (2001) implemented a system that could distinguish between different types of syntactical errors across a variety of online German grammar exercise types, and recommend specific exercises to its users based on the types of mistakes that they made. While these natural language analysis engines have become increasingly complex over the past two decades (see for example Amaral, Meurers, and Ziai, 2011), no non-commercial, research-based ILTS for Dutch has been developed as of yet.

Chapter 3

Project Goals and Design

Having studied the different components of mobile language learning, the goal is now to build an application that blends these together, in an experience that gives the user a highly personalized set of language learning exercises that are generated automatically, but that are in quality indiscriminable from those written by humans. This chapter looks at what such a product might aspire to be: the goals for this project as it was initially conceived, early explorations of features and visual design, and the pedagogical choices that were made. Chapter 4 will focus on the actual implementations of these features over the course of this project, while Chapter 5 will evaluate their implementations and Chapter 6 will evaluate the project as a whole.

3.1 Product specification

The *Language Lion* is an Android application for learning basic Dutch from English. The lion, the national animal of the Netherlands, is featured everywhere from the country's coat of arms to national sports outfits, and the "Language Lion" name is a nod to the application's main inspiration, *Duolingo*, and its cartoon owl *Duo* who provides encouragement to its students.

The application's intended audience is anyone who already speaks English and would like to casually learn Dutch, without any prior experience with the language. This could be in preparation for a vacation to the Netherlands or Belgium, before taking a more intensive language course, or

simply for fun, to gain a basic idea of what the language looks like.

Because the main focus of this project is the technical implementation of the algorithms, and not the design of a complete language course, the end product of this project is not meant to give the learner anything more than a basic introduction to the language. This includes greetings, introducing oneself, and basic vocabulary on food, drinks, and family members. The aim for this project is to bring the learner to a level where they can order simple food and make themselves understandable in a restaurant-like setting. This goal was chosen for its balance between universality and attainability and is one of the pillars of level A1 of the Common European Framework of Reference for Languages (CEFR). Level A1 covers understanding and usage of very basic expressions to satisfy concrete needs, introducing oneself, and asking others about personal details (Council of Europe, 2011). This application aims to span most of the first two goals and the third one to a small extent.

An important requirement is that the app is extensible; that is, after laying these foundations, it should be easy to add new words and phrases to the lexicon, and relatively easy to add new grammatical constructions or to repurpose the codebase for different languages. In addition, it is important that the application's code is completely open-source and free to reuse.

The application is written for the Android operating system. Android was chosen for its ease of access, its open-source nature, and its deep integration with the Java programming language, which allows for easy integration with SimpleNLG. Since this project concerns a proof of technology, and a not an application that will be brought to market or tested on human subjects, certain elements that are usually at the forefront of application development carry less weight here. Pixel-perfect visual design and detailed animations are not a priority, and neither is support for specific assistive technologies such as screen readers. However, the application should be usable, and with clear and comprehensible design and interface patterns. The aim is for the experience to be completely free of bugs, crashes, and freezes, performance needs to be high, and loading needs to either feel instantaneous or be handled through a proper loading screen that provides the user with feedback about the system's status.

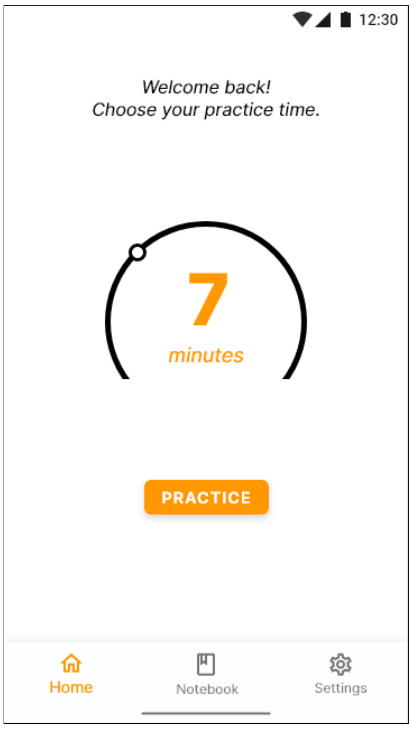

Figure 3.1 Design exploration of the Home screen.

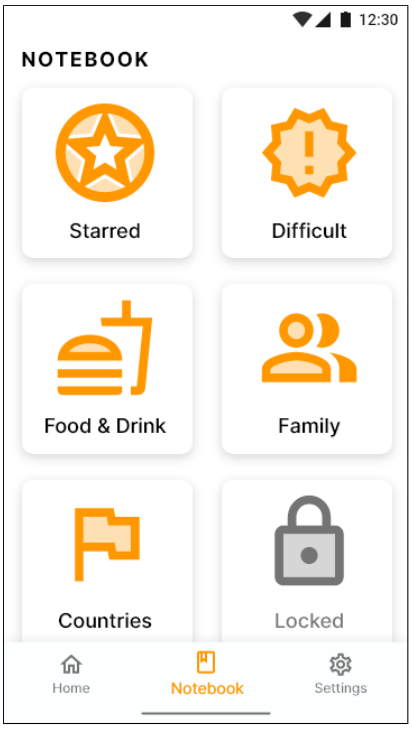

Figure 3.2 Design exploration of the Notebook.

3.2 Proposed feature set

Having established the main goal of the application, we can start to describe its proposed features. Note that the core of this project remains a functioning algorithm for automated exercise generation; many of the features described in this section did not make it into the final project. What is described here is the original vision for the project. Figures 3.1 through 3.4 are design mock-ups, created in Adobe XD, a professional application prototyping tool. They serve as a visual aid to convey some of the ideas listed here, especially the ones that do not appear in the finished product.

The user of the application starts on the Home screen, where they are greeted with a friendly welcome message and a slider that allows them to pick a time between 1 and 20 minutes. After selecting a time, they can tap the PRACTICE button to start a language practice (Figure 3.1).

Since every exercise is automatically generated, the *Language Lion* does not feature themed lessons, as other MLL applications do. Instead, there is an unending stream of exercises for the user to do. As the idea of an infinite number of exercises might seem daunting, and to give the

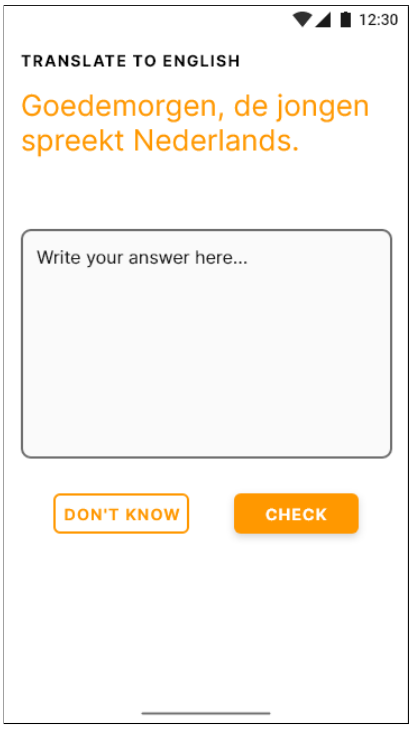

Figure 3.3 Design exploration of an exercise.

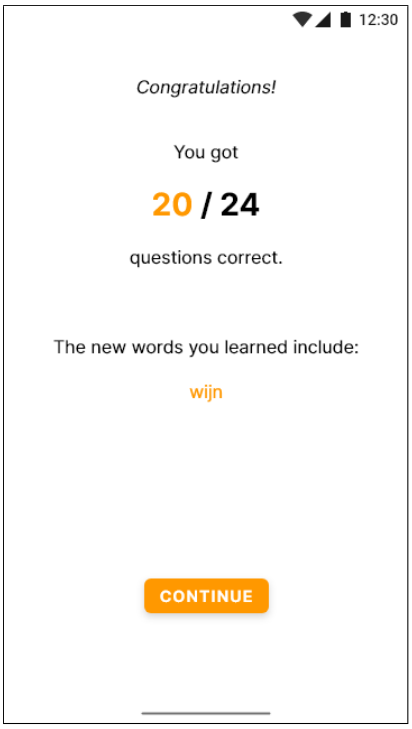

Figure 3.4 Design exploration of the Score screen.

student a feeling of accomplishment after finishing a lesson, as well as a moment of reflection, each practice is constrained to a particular duration. The user can pick how long they want to practice and when the timer runs out, they are automatically transitioned to the Score screen. Picking their own practice time is meant to give the user a sense of control when everything else is being decided by an algorithm and allows them to tailor a practice to their preference. They might feel very motivated and try a 20-minute session, they might be tired and only want to do a quick, 5-minute session, or they might have a daily 8-minute train ride to which this allows them to tailor their practice.

During the practice session (Figure 3.3), the user gets a series of translation exercises, starting with translating Dutch to English and moving to translating English to Dutch when the words in the exercise have already been practiced multiple times. Pure translation, as opposed to fill-in-the-gap or sentence building, is chosen for two reasons. First, it is the most interesting technical challenge, as being able to generate translation exercises requires generation of the other types of

exercises, but not vice versa. Creation of translation exercises necessitates generating all possible translations of a given sentence, whereas a fill-in-the-gap exercise only requires generating answers for the gap, as the rest of the sentence is static. Secondly, it is also the most versatile type of exercise for the student. While translation is a process that they will encounter regularly during natural conversation, especially in early stages of learning a language, similar exercise types are weakly linked to the real world.

To become familiar with the sounds of the language, a text-to-speech engine reads out the Dutch question or correct answer (depending on the question type). Dictation exercises, in which the user has to write down what the text-to-speech engine says, can be used in addition to translation exercises. To practice pronunciation, the student is also able to dictate their answers, rather than type them—this, however, is not a feature of the application per se, but rather a feature already built into most modern software keyboards.

Between exercises, the timer will pause on occasion to show a Grammar Tip. Grammar Tips serve to introduce new grammar in bite-size pieces of text. These tips do not aim to explain an entire grammatical idea at once, but rather to spread them out as new words are introduced. For example, one of the first Grammar Tips would be the conjugation of regular verbs in the present tense, but only for the singular persons. As previous research has advised, tips are kept short to not take the user out of the flow of practices too long, overwhelm them, or distort their practice time (see Section 2.2). Grammar Tips can always be re-read in the Notebook, described below.

After answering an exercise, the user receives positive feedback if they were correct, and the correct answer is displayed if they were not. In addition, the *Language Lion* features an "I don't know" option. When a word is first introduced, it is likely that the user does not know it, but forcing them to answer the question incorrectly can be demotivating. Therefore, the DON'T KNOW button functions as a non-judgmental option that informs the student of the correct answer. Immediately after pressing the DON'T KNOW button, a new exercise with the introduced word is created, to reinforce what the student just learned.

When the timer expires, the user is automatically transitioned to the Score screen (Figure 3.4),

which shows a congratulatory message, the number and ratio of questions answered correctly, and an overview that cycles through the words that were introduced during the lesson. A button will then take them back to the Home screen.

Next to the practice sessions, the user also has access to the Notebook (Figure 3.2). The Notebook is an area of the app which records everything a student has learned. The Notebook is divided into categories based on themes, such as Food & Drink, Family, etc. for easy traversal, and new categories are unlocked as the user progresses through the practice sessions. In addition, there are two user-tailored categories: Starred and Difficult. The Starred section is completely curated by the user, where they are able to 'favorite' items from the other categories that they want to pay particular attention to, and these will then show up in the Starred category. The Difficult section is comparable, but curated by the algorithm, which keeps a running list of the words that the user often makes mistakes in. Both Starred and Difficult items are practiced more often, and if the user gets them correct several times in a row, Difficult items will be removed from the section automatically. Several categories for Grammar Tips, such as Nouns, Verbs, or Word Order, exist as well.

Finally, there is a Settings menu, where the user can manage essential settings such as their account information. There are options present to switch accounts, reset all progress and start from the beginning, and to switch the application to a dark theme to use at night.

3.3 Visual design

Orange is used throughout the *Language Lion* as accent color. Orange is the national color of the Netherlands, after the Dutch royal family's surname, *Van Oranje* ("of Orange", a city in southern France). The application's particular shade of orange, Material Orange 500 or #FF9800, was chosen from Google's Material Design guidelines to make the application fit in well with the rest of the Android operating system (Google Inc., 2020b).

Likewise, all iconography follows the modern Android design standards, and Android's default

typeface, *Roboto*, is used throughout the application. Buttons are either filled with an orange background, if they are the main action on the screen, or outlined with a white background, if they are a secondary action, to draw the user's attention to the main action, such as CHECK ANSWER over DON'T KNOW in a practice.

Chapter 4

Application Architecture and Implementation

The *Language Lion* consists of two separate parts: the Generator, a library that takes in semantic constructs uses them to produce grammatical sentences in Dutch and English simultaneously, and the application itself, which uses the generator to construct exercises to teach Dutch. These components were constructed independently from each other and this chapter discusses both in detail.

4.1 Generating sentences

4.1.1 SimpleNLG-NL

The Generator is a fork of SimpleNLG-NL, as discussed in Section 2.3. SimpleNLG is written in Java and can be imported into any Android application without modification. It is also fully modular, and different modules can be inserted and omitted with relative ease.

At the base of SimpleNLG sits the Lexicon. The Lexicon is a language-specific XML-based database holding every word that SimpleNLG knows, specifying the word's part of speech and any irregularities. An example entry in the default English lexicon is shown in Listing 4.1.

```
<word>
    <id>E0041132</id>
    <base>mouse</base>
    <category>noun</category>
    <plural>mice</plural>
</word>
```

Listing 4.1 An example entry from the default English Lexicon of SimpleNLG.

As Dutch grammar is often less straightforward than English, the Dutch Lexicon includes many Dutch-specific features such as grammatical gender for nouns and perfect auxiliaries for verbs (as Dutch perfect tense can be constructed with either a form of "have" or "be", depending on the verb). SimpleNLG includes both an option to use your own lexicon and built-in default lexicons for Dutch and English. These built-in lexicons are very exhaustive, with several thousand entries each, yet also prone to strange omissions. For example, the Dutch Lexicon did not include the words "drinken" (to drink) and "zien" (to see), but did include entries such as "farao" (pharaoh) and "mee-investeren" (to co-invest). Similarly, of the words that the *Language Lion* teaches, the English Lexicon did not include "strawberry" or "pasta", whereas many less commonly used words were included. Neither lexicon included entries for countries and languages, such as "Dutch" or "English". For this project, these default lexicons were used as a starting point, and entries were manually added as needed.

After selecting a lexicon, the first step in using SimpleNLG is to create a `Lexicon` object, which creates hashed indices of all entries by ID, name, and category for fast and easy retrieval. These Lexicons are then used to create an `NLGFactory` for each language. An `NLGFactory` contains the specifications of several different types of phrases and provides methods to build clauses by populating and then stringing together these phrases. To illustrate the process, an example program in Java is showing in Listing 4.2. We first specify that we are building a clause by creating an instance of `SPhraseSpec`. For simple sentences, we can opt to use shorthand methods to directly set the different parts of speech, such as `clause.setSubject("Ada")` to set the subject on line 7. For more granular control, we can construct individual noun phrases, verb phrases, adjective

```
1  Lexicon lexicon = new XMLLexicon("path-to-lexicon.xml");
2  NLGFactory factory = new NLGFactory(lexicon);
3  Realiser realiser = new Realiser();
4
5  SPhraseSpec clause = factory.createClause();
6
7  clause.setSubject("Ada");
8  clause.setVerb("write");
9
10 NPPhraseSpec complement = factory.createNounPhrase("algorithm");
11 complement.setSpecifier("the");
12 complement.addModifier("first");
13 clause.setComplement(complement);
14
15 clause.setFeature(Feature.TENSE, Tense.PAST);
16
17 String output = realiser.realiseSentence(clause);
```

Listing 4.2 Creating a simple sentence with SimpleNLG, "Ada wrote the first algorithm". The subject and the verb are being set directly to the clause through shorthand methods. For the object, a dedicated noun phrase is created first, to specify its determiner and adjective.

phrases, and prepositional phrases, through their individual specifications.

The factory then builds a tree of different kinds of `NLGElement`. An `NLGElement` can be seen as a "level" of language. At the root of the tree, there is the `DocumentElement`. As its children, it can have one or more `PhraseElement`s that represent clauses, which in their turn have several `PhraseElement`s that represent phrases. Finally, a `PhraseElement` consists of one or more `WordElement`s. When a phrase is created or a new word is added to an existing phrase, the factory looks up the string of text that is passed in the Lexicon and creates a `WordElement` that contains all of the features of the entry. For example, when calling `clause.setVerb("write")` on line 8, the factory looks in the Lexicon and finds the entry for "write" with the `category` of "verb" and two extra features: a `past` of "wrote" and a `pastParticiple` of "written". All of these are registered in the `WordElement`, which is then added to the appropriate place in the sentence tree. Not just `WordElement`s can have features: for example, on line 15, we set the `Tense` feature of the entire clause `PhraseElement` to `Past`, to indicate the sentence needs to be written in the past

tense. Other features include setting negation and passive voice.

As can be seen in the code, when all the elements of a clause have been specified, the next step in the process is to realize the sentence. The `Realiser` (using British spelling, as SimpleNLG was first developed in the UK) takes in the tree and pushes it through five processing modules. Most of the processing is done in the syntax module, which compares the features of all different elements in the tree with each other to determine the proper word order and decide on inflection. For example, in the example in Listing 4.2, the verb "write" receives the features `Person` set to `Third` and `Number` set to `Singular`, courtesy of the subject "Ada", and the feature `Tense` set to `Past` from its parent clause. Meanwhile, in the object of the sentence, the `Modifier` "first" is placed in front of the phrase head "algorithm", by virtue of being of `category` "adjective", whereas an adverb or a prepositional phrase would have been placed after it. When the processing is done, every `PhraseElement` has been turned into a `ListElement` of `InflectedWordElement`s.

Secondly, the tree goes through a morphology module, which applies the features on every `InflectedWordElement` and turns each into a `StringElement`. In our case, since the Lexicon had specified a specific past tense of "write", "wrote", it will retrieve that, and as the English past does not change based on person or number, that is all the processing that needs to be done. Then, every `StringElement` is compared against its direct neighbors in the morphophonology module. While Dutch has no special morphophonological interactions between words, in English this module will change "a" to "an" when required. In fourth place is an orthography module, which provides proper capitalization and interpunction at the end of the sentence and between clauses. Finally, a formatter traverses the tree and returns a normal `String`, in this case, "Ada wrote the first algorithm."

These modules were mostly left alone during the development of the *Language Lion*, as they were already advanced enough for the sentences the app teaches, with one exception. SimpleNLG allows for adding *front modifiers* to clauses, such as "By the way" or "Excuse me". While the Dutch orthography module properly added a comma between a front modifier and the rest of the clause, the English orthography module did not, and was therefore adapted to provide this

functionality as well.

In addition, while the Generator was originally compiled for Java 11, to ensure compatibility with the Android system and the Android Studio IDE it had to be recompiled into Java 8. Thankfully, this never caused any major issues to the code base.

4.1.2 The Semanticon

While SimpleNLG does a good job of building sentences, it requires its input to be in the target language, and we need a common base to be able to generate the same sentence in two languages at the same time. Enter the idea of the *Sememe*, a concept original and unique to this project[2]. The assertion behind Sememes is that humans that speak similar languages or are part of similar cultures, also share a conceptual understanding of many things in the world. While this is impossible to prove, as the mind remains as inaccessible as ever, it should not be too controversial to assert that most Dutch and English speakers share the same idea of what milk is, even though English speakers call it "milk" and Dutch speakers call it "melk". For our purposes, the concept of milk, the Platonic ideal, if you will, is called a Sememe, or semantic element.

Of course, this won't hold for each concept, as each language and culture has concepts that are unique to itself. For example, a native Dutch speaker would have a hard time translating the adjective "gezellig" to English, usually resorting to alternatives such as "cozy" or "with a good atmosphere" that only ever seem to partially cover the meaning of the word. However, this is an introductory language course, not advanced translation studies, which means that these nuances can often be disregarded as they are simply not taught. For basic language teaching, it turns out, the Sememe is a very useful idea.

Sememes can be encoded in a Semanticon just like lexemes are encoded in a Lexicon. An example is shown in Listing 4.3. Most sememes are named after their English symbol, as more people speak English. Sememe IDs are strings of eight characters: the first is "S", to distinguish

[2]While the study of structural semiotics formalizes the "sememe" as an atomic unit of meaning analogous to the syntactical morpheme, that is not what is studied or meant here.

```
<sem>
    <id>S0041016</id>
    <name>milk</name>
    <english>milk</english>
    <dutch>melk</dutch>
    <categories>
        <drink_item />
        <no_determiner />
    </categories>
</sem>
```

Listing 4.3 An example entry from the Semanticon that was constructed for this project.

them from lexemes labeled "E" (for entry), the first three numbers define the general part of speech (noun, verb, adjective, etc.), and the last four numbers form the item ID within their category. The English and Dutch tags contain all symbols linked to the sememe for either language, and can contain a list of symbols: for example, the "greeting_quick" sememe contains the symbols "hi" and "hey" for English, and "hoi" for Dutch. Lastly, the sememe can be part of any number of specific categories, which are used by the application's Grammar module to create sentences, as will be outlined in Section 4.2.5. The full contents of the Semanticon are listed in Appendix B.1.

The biggest strengths and limitations of the Semanticon become evident when we consider homonyms. While many native speakers might never have thought about this, the English word "live" actually carries two meanings: it could describe a location, as in "I live in New York", or it could simply mean that its subject is alive, as in "the boy who lived". While more or less related, these are two different concepts, and the Dutch language has two different verbs for these: "wonen" and "leven", respectively. As such, the Semanticon contains two different sememes: "live_in" corresponds to "live" and "wonen" while "live_alive" corresponds to "live" and "leven".

A second big strength of the Semanticon lies in its ability to quickly generate multiple correct sentences. Since the contents of the English and Dutch tags can be a list, it is easy to iterate over all options. For example, a sentence with "greeting_quick" as front modifier, "you" as subject, "be" as verb, and "good" as complement would generate into "Hi, you are good" and "Hey, you are good" in English, and "Hoi, jij bent goed", "Hoi, je bent goed", "Hoi, u bent goed", and "Hoi,

jullie zijn goed" in Dutch, which has both emphasized ("jij"), de-emphasized ("je"), polite ("u"), and plural ("jullie") versions of the second-person pronoun.

Implementing the Semanticon mirrors the implementation of the Lexicon in that a `Semanticon` object is constructed which indexes all sememes by ID, name, and category. These are all stored as a `SemElement`, a new, even lower-level type of `NLGElement`. Each of the words in the Dutch and English tags of the sememe then corresponds to an entry in its language's lexicon, so by iterating over all of them, a clause for each can be created, and we end up with a few sentences in each language that correctly translate into each other.

It must be noted, however, that these strengths become weaknesses when it is not possible to determine from the context of the sentence which concept is meant. For example, if a student is tasked with translating just "I live", this could be either "ik woon" or "ik leef", but since we are iterating over just one sememe, only one of the options will be generated. Thankfully, this can be easily remedied by simply not generating sentences that could be ambiguous.

Before moving on, it is worth it to take one more look at how personal pronouns function within the Semanticon, as some attention needs to be paid in order to make them work correctly. In the case of the second person, while theoretically the Dutch singular, plural, and polite versions of the pronoun should constitute different sememes, they are all put together as different options of one "you"-sememe, in order to allow all correct translations to be generated. This works especially because in English, there is no difference between the singular and plural second-person form of a verb: "you are" can be both singular and plural. The opposite is true for the Dutch pronoun "zij" and its de-emphasized variant "ze", which translate to both "she" and "they". In this case, they are listed as different sememes, because the Dutch verb *will* change based on number. Here, it is always clear from context which translation is meant: "zij/ze is" translates to "she is", and "zij/ze zijn" translates to "they are". This just illustrates once more that while it is easy to add new members of open word classes to the Semanticon, such as nouns or verbs, careful curation and testing is required to make sure all members of closed classes such as pronouns work properly.

4.2 Writing for Android

The Generator library is fully written in Java and can therefore be imported into any Android application as a Java Archive just like it would be in any other JVM-based project. What is left, then, is making use of the library by building an actual application.

4.2.1 Overview

While Android applications have traditionally been written in Java, support for the more modern Kotlin language was added in 2017, and only two years later Google announced that Kotlin would be the preferred language going forward (Cleron, 2017; Haase, 2019). Kotlin runs on the Java Virtual Machine and is completely interoperable with Java, but includes many quality-of-life improvements, such as more concise syntax, better support for higher-order functions, named parameters and default arguments for functions, and better type safety with support for non-nullable types. For these reasons, the *Language Lion* has been completely written in Kotlin.

Android apps are built up of Activities, core Android classes that draw the user interface and handle user input. Activities themselves consist of Fragments, which can be thought of as representing the different "pages" of an app, each representing a portion of the UI in an activity. The *Language Lion* consists of only one activity, given that it only has one "environment" for the user to be in, and this activity consists of five fragments: Home, Lesson, Score, Notebook, and Settings.

The Home fragment (Figure 4.1) is the app's start destination and the simplest fragment. Staying within the application's theme of random assignment, it randomly displays one of three friendly and motivating welcome messages: "A lesson a day keeps the boredom away", "Welcome back, Rayo! Keep up the good work :)", or "Your Dutch lessons won't take themselves!". It is not possible to change the name displayed in the second message, as this was deemed an inessential feature, and the developer of this project does not mind his name to be displayed in more apps. Below the greeting, the user can start practice by tapping one of three timer buttons: one minute,

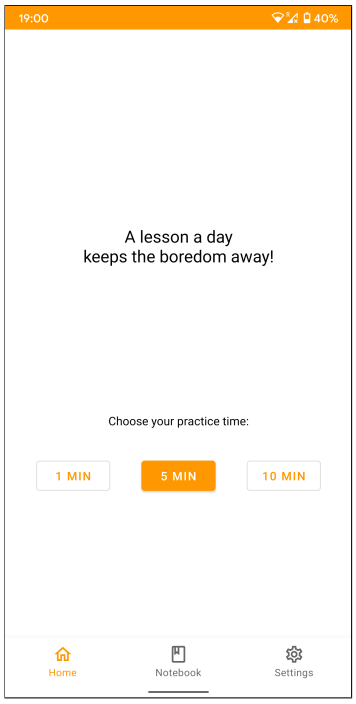

Figure 4.1 The home screen of the *Language Lion*.

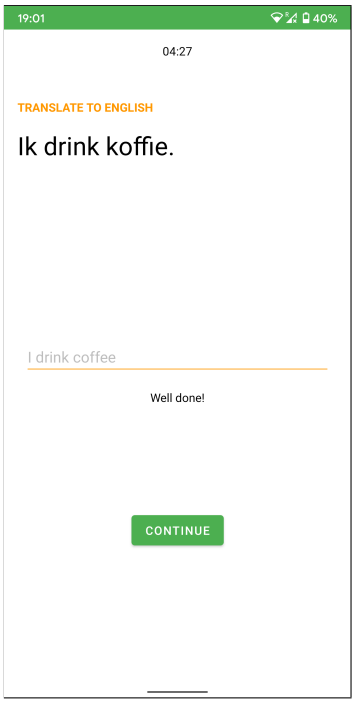

Figure 4.2 Positive feedback for a correct answer.

five minutes, or ten minutes. This is a simpler version of the custom control originally pictured in the design explorations (see Figure 3.1), as designing a custom UI control was, again, deemed inessential to the core functionality of the app. The one-minute option is available mainly for demonstration purposes, but the choice between five and ten minutes should provide at least some flexibility to testers.

The other two top-level destinations, Notebook and Settings, are available through the bottom navigation bar, which is implemented following Google's standard Material Design guidelines (Google Inc., 2020a). The Notebook functions as a basic list of all sememes that have been learned so far, displaying their corresponding Dutch words on the left and English words on the right side of the screen. The Settings screen is light on options, as the application is not meant for widespread use, but it does contain functionality to reset all progress and to switch to a pre-configured profile where some exercises have already been completed, for easy testing. Both fragments are explored in further detail in Section 4.2.6.

Most resources have been dedicated to the Lesson fragment, as is to be expected. It displays

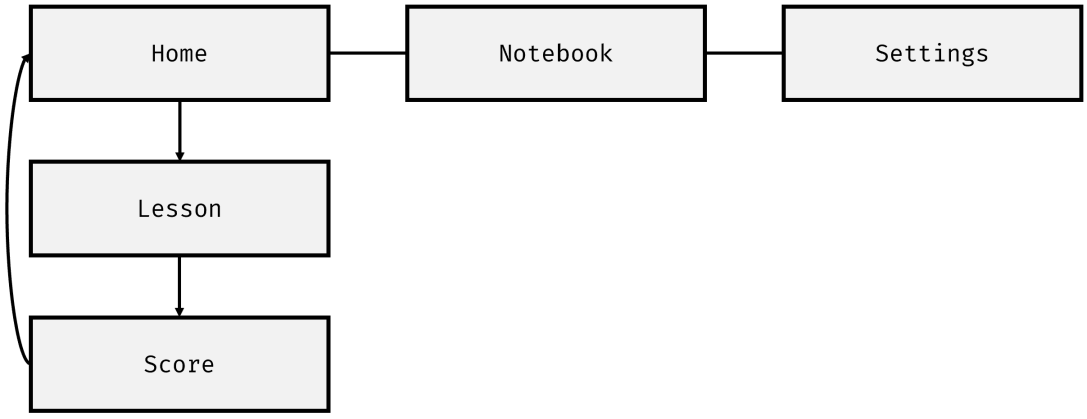

Figure 4.3 The navigation graph of the *Language Lion*.

a timer, starting at the amount the user specified, in the top, and the exercise below, with two buttons: CHECK ANSWER and DON'T KNOW. CHECK ANSWER will check the answer and provide feedback. If correct, it colors the application UI green and displays a randomly generated congratulatory message such as "Excellent!" or "Well done!". If incorrect, it colors the application UI red and displays (one of) the correct answer(s) to the question. When the DON'T KNOW button is pressed, the correct answer is also given, but the application UI stays orange to not give the user the feeling they are being penalized. The entire process is explained in more depth in sections 4.2.4 and 4.2.5. Regardless of the outcome, new questions will be generated until the timer runs out, and the user is automatically transitioned to the Score screen, which displays the number of exercises they got correct and displays a button to return to the Home fragment.

While the application targets the newest version of Android, Android 10, it is compatible with devices starting from Android 6.0 or higher, corresponding to API level 23. Its code is available on GitHub under the MIT license, which permits all modification and distribution of the code for any private or commercial use. Appendix A features instructions on how to install the application from the GitHub page.

4.2.2 Fragment architecture

Android development encourages following the Model-View-ViewModel (MVVM) architectural pattern, which was therefore implemented in the *Language Lion* as well. It means that every fragment consists of four separate files. The layout, or view, is specified in XML. The ViewModel is a class that holds the data associated with the fragment, and the methods that act on that data. For example, in the Lesson fragment, the ViewModel holds the current exercise, the methods to check the answer, the methods to build a new sentence, et cetera. `ViewModel` objects are instantiated through a `ViewModelFactory` which allows for passing arguments such as the selected timer coming into the Lesson or the score to be displayed on the Score screen. Lastly, connecting the View and the ViewModel is the UI controller, a subclass of `Fragment` that displays views and handles user input.

When a fragment is navigated to, the UI controller creates a `binding` object by *inflating* the View, or putting all the elements of the view in memory for easy access, and then constructs the corresponding ViewModel through the `ViewModelFactory`. After this, we can specify *listeners* for each UI element and ViewModel variable, allowing the UI controller to know when the listener's subject is changed, for example a button that is pressed, or a timer that runs out. This creates a constant back-and-forth between the UI controller and the ViewModel. For example, when a user presses the "check answer" button in the Lesson, several things happen:

1. The UI controller registers that the user presses the button;
2. The UI controller reads the text in the answer text field;
3. The UI controller calls the `onCheck()` method from the ViewModel and passes the answer text as an argument;
4. The ViewModel checks whether the answer is correct (through logic described in Section 4.2.5) and, when finished, updates its `LessonStatus` value from `INPROGRESS` to either `CORRECT` or `INCORRECT` (all custom enumerated values);
5. The UI controller, which has a listener for `LessonStatus`, is notified that the variable is changed;

6. Depending on the value of `LessonStatus`, the UI controller makes some changes to the UI: it disables the answer field so it cannot be changed anymore, it hides the "check answer" and "don't know" buttons but makes the CONTINUE button visible, and it changes the color of some of the UI elements to green or red, depending on whether the answer was correct.

Then, when the CONTINUE button is pressed, `LessonStatus` is reverted back to `INPROGRESS`, causing the UI elements to change back to orange, et cetera. While it may seem tedious to constantly move between two classes, this separation of concerns is important, as UI controllers are recreated during every configuration change, such as screen rotation or changes in keyboard availability. The ViewModel, however is not. Not following this pattern would for example mean that in the Lesson fragment, the current question is reset every time the software keyboard is opened or closed, which would make it impossible to correctly complete an exercise.

4.2.3 Database

Android applications can access a SQLite database through the Room persistence library. The database of the *Language Lion* is divided up in three tables: Sememes, Sentences, and Profiles. Their specifications are listed in Table 4.1.

The Sememes table contains a list of all sememes that are taught in the application. It is different from the Semanticon as described in Section 4.1.2 in that the Semanticon stores the semantic data associated with the sememe, such as its associated Dutch and English lexemes, whereas the database table stores the pedagogic data associated with the sememe. While the Semanticon is initialized every time a lesson starts, the database persists between app sessions. The table's primary key is the `sememeID`, which is the same eight-character string as specified in the Semanticon. The `learned` field starts out as `false` and is set to `true` when the student has gotten at least one exercise with the sememe in question correct. The `stage` field tracks how many times the user has gotten an exercise with the sememe correct, up to 8. This value is used in determining the type of exercise and the spaced repetition, as is explained below.

The Sentences table lists all the sentence templates (see Section 4.2.5) and tracks whether they

Sememes	
sememeID	String
learned	Int*
stage	Int

Sentences	
sentenceID	String
learned	Int*

Profiles	
profileID	Int
username	String
primaryQueue	LinkedList<String>
secondaryQueue	LinkedList<String>

Table 4.1 The *Language Lion* database schema. Int* indicates that the type is actually Boolean; as SQLite does not support Booleans, they are stored as Integers that are either 0 or 1.

have been learned or not in a similar way to the Sememes table. Again, this table only contains their ID and `learned` value, while the Lesson fragment contains its own Grammar index with more detailed template information.

As the application does not support multiple users, but the learner's progress does need to be stored, the Profiles table consists of only one entry: a profile with an ID of 0, a `username` of "Rayo" (which ended up not being used, but remains present for compatibility) and the full primary and secondary learning queues stored as a Java `LinkedList` each (see Section 4.2.4). Because SQLite does not support storing data as custom Java objects, the database implements a type converter which translates the `LinkedList` to a string of JSON and back every time either value is accessed, using Google's open-source Gson library.

In Android projects, databases are fully written through Kotlin classes as well. Each table is specified as a separate data class of which every value represents a column. Java annotations such as `@PrimaryKey` and `@ColumnInfo` are then used to specify the function of each column. Queries are specified in a separate class, a *data access object* or DAO, which is an interface where every abstract function is annotated as a `@Query`. For example,

```
@Query("SELECT * FROM sememe_table WHERE sememeId = :key")
fun getSememe(key: String): Sememe?
```

allows us to query the Sememes table to get a specific sememe by calling the `getSememe()` function from a ViewModel as if it were any other method. (The "?" at the end of the return type

declaration indicates that the type is nullable.) Insertion, updating, and deletion work similarly.

Another Android-specific feature is that by default, database access on the main thread is not allowed. Therefore, every ViewModel that interacts with the database should include a job scheduler and calls to the database should only be made from a secondary thread. The *Language Lion* adheres to these guidelines for the sake of code quality, though calls to the database are limited enough that they would not have a noticeable performance impact when run on the main thread.

4.2.4 Encoding pedagogy and tracking progress

So far, this has been a rather technical look at the application's architecture and some of the basic patterns of Android development, but hopefully, examining how the different classes interact with each other and with the database has given some insight into what it takes for the *Language Lion* to function. With that out of the way, the next two sections look exclusively at the Lesson fragment.

As can be seen in the database, the learner's profile has two queues associated with it, one primary and one secondary. Both are implemented using a `LinkedList` from the `java.util` library. When using the application for the first time, the primary queue contains the ID of every sememe that the app teaches in the order that they will be introduced. The first sememe that is introduced is "to be", followed by "to eat", "to drink", "apple", "banana", and so forth. The full ordering is listed in Appendix B.3.

Every time a new exercise is created, the head of the queue is popped off, and that sememe is used to build the exercise, through the process that is described in the next section. If the exercise is answered incorrectly or the "I don't know" option is picked, a new exercise is created using the same sememe, until a correct answer is given. If the exercise is answered correctly, two things happen: the sememe's `stage` is increased by one and it is reinserted at a later point in the queue.

A sememe's `stage` starts at 0 and can increased up to and including 8. When a sememe is popped off the queue to build an exercise, its stage determines what kind of exercise is built: if

the stage is below 4, the task is to translate a Dutch sentence to English, and if the stage is 4 or higher, the task is to translate an English sentence to Dutch, which is generally more difficult. In addition, when an exercise is answered correctly, the sememe is added into the queue at position (`stage` + 2). For example, when a sememe has been correctly used in an exercise for the first time, it will be reinserted in the queue at index 3. The queue uses zero-based indexing, so a new exercise with the sememe will occur after three others. (Note that the need to insert sememes at specific indices is also the reason why the queue is not, technically speaking, a `Queue`, but rather a `LinkedList`. Because the list is still first-out-only, however, it is easier to picture it as a queue.)

The maximum of 8 stages was chosen to make the student go through several exercises in both directions of translation before the maximum is reached, and roughly mirrors word progression in the *Memrise* application, described in Section 2.1. When a sememe reaches stage 8, it is not reinserted into the primary queue, but rather onto the end of the secondary queue. In the application's current form, the secondary queue has only very limited functionality, but the idea is that as the *Language Lion* grows, it would handle the spacing and reintroduction of sememes that have been learned in the past. Currently, when a sentence is generated, the sememes that are not the main sememe are picked randomly; for example, if the generation algorithm needs a personal pronoun, it will randomly choose from the personal pronouns that have been encountered before, without much regard for the specifics of those encounters. Future versions of the application could use the secondary queue to more systematically cycle through different options, in this case finding the first personal pronoun that occurs in the list to use, and then reinserting it at the end when it has been used.

It is important to remember, however, that the popped-off-the-queue sememe, while the main subject of the exercise, will never be the only sememe tested in an exercise. For instance, if the head of the queue is "coffee", and the exercise generated is to translate the sentence "We drink coffee", then the sememes "We" and "coffee" are also tested. To account for this, when an exercise has been answered correctly, the stages for *every* sememe are increased by one, and not just the main one. What this means in practice, is that words that are repeated often, such as personal

pronouns and common verbs like “be” and “eat”, reach stage 8 very quickly and are moved to the secondary queue first. This is useful, since it prevents these sememes from cluttering the primary queue with ‘useless’ exercises. For example, since different items of food take up a sizable portion of the sememes that are being taught in the *Language Lion*, and the verb “to eat” will often be chosen to accompany them, the student probably does not need separate exercises to go over the sememe “to eat”.

This is the way pedagogy is encoded and enforced. Re-introducing the sememes at progressively later stages in the queue ensures that they are still regularly practiced while new sememes can be introduced gradually, as is evaluated in further detail in Section 5.2. As most of the resources available for this project were used in building the sentence generation algorithms, the pedagogy ultimately became somewhat of a secondary priority, and is hardly as advanced as some of the options explored in Section 2.5. However, given the inability to test on human subjects, detailed student models were never going to be able to be properly implemented and thoroughly tested on this initial version, and focusing on language generation instead was probably the wise choice.

4.2.5 Building exercises

That said, it is time to bring all elements together and look at the final piece of the puzzle. On the one hand, there is the Generator, which realizes a sentence after specifying its different components, and on the other hand, there is the Queue, which provides a sememe that the exercise needs to be built around.

As it turns out, the first hurdle to overcome is to simply get the Generator to work in an Android environment. The archive itself, after being compiled for Java 8, can be imported with ease, but the Semanticon and English and Dutch lexicon XML file have to be supplied externally. As this is external, read-only data, they can only be stored in the `assets` directory of the Android app. The contents of this folder are then packaged together with the code binaries in the app’s APK files. This also means that there is no absolute path to these files, however, which becomes problematic, since SimpleNLG requires the URI of a file in order to use it to initialize the Semanticon and

```
<sentence>
    <id>G003</id>
    <subject>person</subject>
    <verb>verb_drink</verb>
    <complement>drink_item</complement>
</sentence>
```

Listing 4.4 An example template from the Grammar that was constructed for this project.

Lexicons.

There is no elegant way to solve this without completely breaking up the Generator code, which would have given a considerable overhead to the project. Instead, after considering multiple options, the cleanest solution in terms of code was to copy the XML files from `assets` to the application cache upon the start of a lesson, to generate a URI from there. This introduces a noticeable performance impact when loading a new lesson, but alternatives, such as duplicating the XML parser in the Android code to create a temporary file, were found to be even more impactful.

With the Semanticon and Lexicons set up, there is one more asset required to build exercises, which is the Grammar. The Grammar is a context-free grammar that consists of a series of sentence templates that are used to construct exercises. It is encoded in XML as well, as can be seen in Listing 4.4. Each template has a four-character ID, a `subject`, a `verb`, and an optional `complement`. Each (non-ID) field contains a string that corresponds to one of the categories that can be given to the sememes in the Semanticon. For example, the `person` category is given to all personal pronouns and nouns that can describe a person, such as “man”, “woman”, “child”, et cetera. The `verb_drink` category is given to all verbs that can link a subject to a drink complement, such as “be”, “have”, and “drink”, and so forth. The full Grammar is listed in Appendix B.2.

Deciding on what categories to use is to find a compromise between correctly representing a sememe’s real-world use and not getting lost in the details. The easiest way to build a context-free grammar would be to simply match a word’s part of speech to a slot in the sentence, with templates such as `<noun><verb><noun>` or `<pronoun><noun><adverb><adjective>`. However,

this will cause massive overgeneration of nonsensical phrases, such as “the apple speaks a book” or “we jump fairly delicious”. On the other hand, creating detailed categories for each sememe individually by hand would require so much labor that any gains from automation are lost. The templates in the Grammar of the *Language Lion* aim to strike a balance. Some overgeneration is still present, mainly since all personal pronouns and nouns describing people are taken together in the `person` category. For example, the template in Listing 4.4 will generate not only “it is wine” but also “I am wine” or “she is wine”. While most will not use these sentences on a regular basis, they can still be useful as exercises, as taking words slightly out of context will encourage thinking deeper about their use, and any humorous combinations might even be remembered better than regular sentences.

At the start of a practice session, all sentence templates are read in and indexed based on ID and containing categories, for easy retrieval of all templates that feature e.g. a `person` or `food_item`. Then, when a new exercise is created, and a sememe is popped off the queue, all previously-encountered sentence types (stored as `learned` in the database similarly to sememes) that feature one of the sememe’s categories are retrieved. One of these is then picked at random; if there are no templates available, because the student has not encountered a sememe of its specific categories before, a new sentence template is unlocked and its `learned` value is set to `true`.

With a template ready, for each position (subject, verb, or complement), it is tested whether the category specified in the template matches one of the sememe’s categories. If it does, that sememe will be used for that position, and random fitting sememes are picked for the other positions.

Now, for each noun phrase with a noun as its head (and not a pronoun), we need to decide on a determiner, as the Generator does not add these by default. At random, either “the” or “a” is selected. While both are sememes in the Semanticon, they are never part of the queue, but rather unlocked from the start. The Dutch definite article distinguishes between common nouns, which receive “de”, and neuter nouns, which receive “het”. As all nouns already have their gender encoded in the Lexicon, we can decide on which one to use by simply looking up the gender while building the sentence, later.

However, what is not encoded in the Lexicon is whether the noun is countable or not. Many of the words that the *Language Lion* teaches are uncountable, specifically drinks such as "water" and "milk". To work around this, mass sememes get the extra category `no_determiner` (as featured in Listing 4.3). While technically a violation of separating the levels of language—as this is a syntactical feature, and not a semantic one—for the words that are taught in this course, their countability is the same between English and Dutch, which makes it easier to build it into the Semanticon rather than opening up the SimpleNLG code. The result is the same: when a noun phrase gets assigned the indefinite article, but the sememe is of category `no_determiner`, the article is omitted.

When the chosen subject happens to be the sememe "they", this has to be registered as well. As noted at the end of Section 4.1.2, the Dutch lexeme "zij" means both "she" and "they". When "zij" is passed to SimpleNLG, however, it will simply default to assuming it means "she", singular. To correct this, if the specific "they" sememe is encountered, this is encoded in a special Boolean value `pluralZij`.

Lastly, before moving on, there is one more special category of sememe, which is `interjection`. Interjections, such as "Hi", "Good morning", "Excuse me", or "Cheers", are taught starting in the second half of the course. As these interjections can conceivably be added to the front of any sentence, there are no templates that include an `interjection` field; instead, when the to-be-practiced sememe is an interjection, it is simply added to the front of a completely random sentence.

Taking all of these factors into account, there are two calls to `buildClauses()`. This method takes in a language (Dutch or English); a subject, verb, optional complement and optional interjection as lists of strings that correspond to the `dutch` and `english` fields of their sememes; a few booleans that describe which determiners should be used for any noun phrases; the boolean `pluralZij`, and finally the language's specific `NLGFactory`. This method iterates over all possible options of subject, verb, complement, and interjection, setting specific features where needed, and returns a set of clauses that will translate correctly into the set of clauses returned by the function

call for the other language. A full evaluation of this process is detailed in Section 5.1. For now, we map over every clause to pass it through the `Realiser`, and then, depending on the main sememe's stage, pick one sentence from either the Dutch or English set to be displayed as the exercise. The other set of clauses forms the set of all possible answers to the question.

When a user answers the question, orthography is not of importance, so before comparing the input to the set of answers, both are stripped of capitalization, interpunction, and trailing or double spaces. This is the entire process. More sememes, as well as more variations on the subject-verb-complement pattern, can be added to the current Semanticon and Grammar without any need to modify the current code and they would work perfectly, and more difficult grammatical patterns can be added with relative ease by adding more permutations to the code described so far.

One more thing of note about the Lesson is that the underlying database is updated every time an answer is correct, rather than at the end of the lesson. This means that while the Score screen is only displayed when the timer runs out, a user can close the app or use the back button at any time during a lesson without losing their progress.

4.2.6 Notebook and Settings

Still left to be discussed briefly are the Notebook and Settings sections of the *Language Lion*. As neither section is essential for the application to function, and user testing was, as said, not possible, they are both more starting points of vision than fully implemented areas. That said, they are there, and they are functional.

The Notebook section displays every sememe which has been encountered, meaning their `learned` value has been set to `true`. When using the app for the first time, it already includes entries for "the", "a", and "I", which are needed to make the first few exercises function. All sememes are grouped by category: first interjections, then articles, then personal pronouns, verbs, family nouns, food nouns, drink nouns, and finally adjectives. It is implemented using Android's `RecyclerView`, which is optimized for large lists of items, and dynamically adds and removes items

Figure 4.4 The Notebook screen.

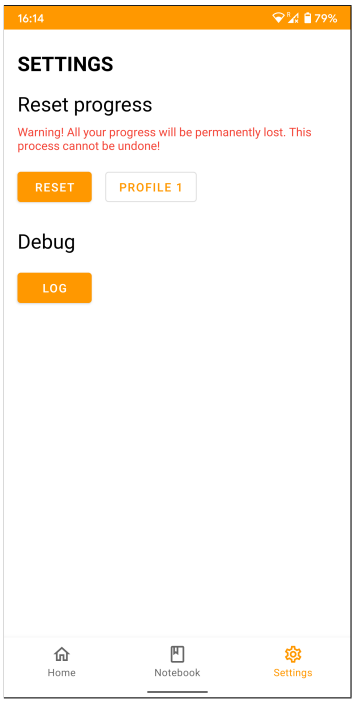

Figure 4.5 The Settings screen.

from the list as the user scrolls through it and they enter and remove the screen. To make this work, a custom-written `SememeAdapter` adapts every learned sememe in the database to a child of the `RecyclerView`.

Since the Notebook lists every Sememe, the different Dutch versions of "you" are all listed under one entry, as discussed in Section 4.1.2. In a fully worked-out version of the Notebook, these should be split up, preferably with added tips and notes on grammar.

As the app does not have users with preferences, the only option necessary on the Settings screen is to reset all progress to the very start of the Queue. For testing purposes, a pre-initialized profile is included as well, and can be accessed by tapping the PROFILE 1 button. Using this profile brings the user to a stage in learning after completing the first 125 exercises, or when the sememe "coffee" has just been introduced. This is equivalent to the developer of this app typing as fast as possible during one ten-minute learning session from the start of the learning process. The LOG button exists for testing purposes as well and will log the entire contents of the database to the Android Studio console for debugging.

4.3 Complete pedagogy

Now that we understand how to build exercises, the last question left to answer is which exercises to build, and how many of them. How and when the Queue reinserts sememes has already been discussed in Section 4.2.4; instead, this section is focused on the considerations going into what sememes and sentence templates are taught, and in what order.

The complete Semanticon is listed in Appendix B.1 and the complete Grammar in Appendix B.2. The course has implemented 78 sememes and 10 sentence templates in total, and 16 sememe categories, including the special case `no_determiner`. When accounting for each possible permutation, including the difference between definite and indefinite articles, the *Language Lion* can theoretically generate 3346 different exercises. This means that, on average, there are 43.5 exercises to every sememe—a distorted figure, since based on their part of speech the sememes' usages vary wildly—but nonetheless a high number.

The order in which new sememes are introduced (listed in Appendix B.3) can be divided up into several blocks. The first three sememes are "to be", "to eat", and "to drink", since they form the basis of all food-related conversation. After this, the first block is focused on teaching the 15 food and 7 drink items, selected for their universality and early appearance in the *Naar Nederland* textbook. While the course starts with the sememe "I" unlocked, as the student is progressing through this block, the other nominative personal pronouns are unlocked in order as well. Near the start of the block, "good" and "bad" are taught as well, and "tasty" is added at the very end of the block, to teach the student basic comments about food. Block 2 is very short and teaches the verbs "to have" and "to speak" and the languages "Dutch" and "English", which are deemed essential. Then, block 3 alternates between introducing basic terms to describe people, such as "man", "woman", and "child", with introducing more basic verbs, such as "to go" and "to walk". Finally, block 4 alternates between introducing 16 interjections ("Hi", "Hello", and so forth) with first the accusative personal pronouns, and later the verb "to read" and a few readable nouns such as "menu" and "book".

These sememes prepare the student for basic ordering in a restaurant; really, the only sememes

that are missing are some modal verbs such as “can” and “may”, the numbers, and some restaurant-related sememes like “to pay” and “bill”. Due to the extensibility of the platform, however, these can all be implemented easily. Grammatical features that would be needed include plural nouns, interrogative sentences, and negation, but as all of these are already supported by the Generator, the implementation is just a question of writing and testing the appropriate sentence templates in the Grammar. Other grammatical features have other restrictions: for example, the course does not teach gender neutral pronouns such as the singular they, as the Dutch language (disappointingly) does not have a standardized counterpart (and, as has already been discussed in length, using “zij” would only cause confusion).

Chapter 5

Evaluation

The project was evaluated on three separate levels. First, the ability of the Generator to generate all correct sentences using the Semanticon and Grammar of the app; second, the ability of the Queue to properly introduce and revise all Sememes; and lastly, the application as a whole, its performance, and its user interface. Here, it was of importance to be able to isolate each component and find the best way to evaluate each one individually before putting them all together.

5.1 Generating sentences

In order to test the combination of SimpleNLG-NL, the Semanticon, and the Grammar to produce correct translations, the relevant code was first written in a separate Kotlin project before adding it to the Android app. Instead of retrieving a sememe from the queue and basing an exercise on that, for each of the sentence templates in the Grammar, random sememes were picked to fill its slots and all generated possibilities were printed. Each sentence received a randomly chosen interjection as well. For example, sentence template G004 (see Listing 4.4) could get the sememes "you_sub" as subject, "drink" as verb, and "milk" as complement, with "a" as the determiner for the complement, and "greeting_quick" as interjection, to generate the following sentences:

- Hi, you drink milk.
- Hey, you drink milk.
- Hoi, jij drinkt melk.
- Hoi, je drinkt melk.
- Hoi, u drinkt melk.
- Hoi, jullie drinken melk.

As there are currently ten templates in the grammar (see Appendix B.2), each run of the code would generate ten of these sets of sentences, each time with different sememes, and after each run the grammatical correctness of each generated sentence was evaluated.

Since Android applications usually took roughly a minute to build on the development system, writing the code in a separate project first allowed for much faster iteration on problems such as accounting for the countability of nouns and the double meaning of "zij" (see Section 4.2.5). It also allowed for the quick identification of bugs in SimpleNLG-NL, such as the absence of a comma after front-modifiers in English sentences (see Section 4.1.1).

In addition, seeing all permutations of a sentence printed out makes clear which grammatical features are and are not supported. As can be seen in the above example, all possibilities of the sememes "greeting_quick" and "you" are generated, for each version of the Dutch "you" the verb is inflected correctly, the determiner "a" is omitted as "milk" has is of category `no_determiner` and the sentence is formatted correctly. What can also be seen, however, is that the Generator does not include support for continuous verb forms ("you are drinking milk"). While Dutch includes continuous forms as well ("jij bent aan het drinken", lit. "you are at the drinking"), in Dutch, they can be used interchangeably with the simple present, with more flexibility than in English. "You are drinking milk" would generally be considered a correct translation of the Dutch "jij drinkt melk", while "you drink milk" would not be a correct translation of "jij bent melk aan het drinken". As the Generator does not distinguish between the source and target languages, these sentences cannot be properly generated, and the present continuous is not supported, even though these translations are correct.

Another major grammatical feature that is not supported are contracted forms in English, i.e. "I'm" and "you're" are not considered valid translations of "ik ben" and "jij bent". While SimpleNLG's morphophonology module can update words based on their direct neighbors, each

word is still treated as its own distinct unit, and merging words into contracted forms is not currently possible.

Thorough testing surfaced one more perplexing bug that has not been fixed yet. When the accusative personal pronoun "her" is set as the complement of the sentence, the English realizer will change the word into "hers", generating sentences such as "I saw hers". It does not do this for Dutch, it does not do this for other accusative personal pronouns such as "me" or "his", and the lexicon entry for "her" is not any different from those of other pronouns. As accusative personal pronouns were only added in a relatively late stage of development, this is the only actual bug in the Generator that is still present in the current version of the *Language Lion*.

5.2 Sememe introduction and spaced repetition

Since testing on human subjects was not possible, trying to implement and fine-tune a highly personalized spaced repetition algorithm was not a valid option. That did not mean that the Queue did not need to be tested, however, as it was still important to verify that 1) all sememes would get properly introduced over time, 2) all sememes would be repeated a few times before too many new sememes were introduced, and 3) all sememes that are not common grammatical sememes such as the personal pronouns would increase their stages at roughly the same pace.

While anecdotal evidence from testing the application during development had already suggested that these requirements were met, for more rigorous testing, data was recorded from taking four five-minute practices, from the very beginning of the course. After every practice, for every sememe that had been introduced at that point, its `stage` was recorded and plotted in Figure 5.1. A sememe's stage is a value from 0 to 8 that increases every time an exercise containing that sememe has been answered correctly, as described in Section 4.2.4. These practice sessions were taken by the developer, who is already a native speaker of Dutch, and are therefore not representative of the speed at which a new learner would learn new sememes (as they would likely think longer about each exercise and make more mistakes). It is, however, representative of the relative

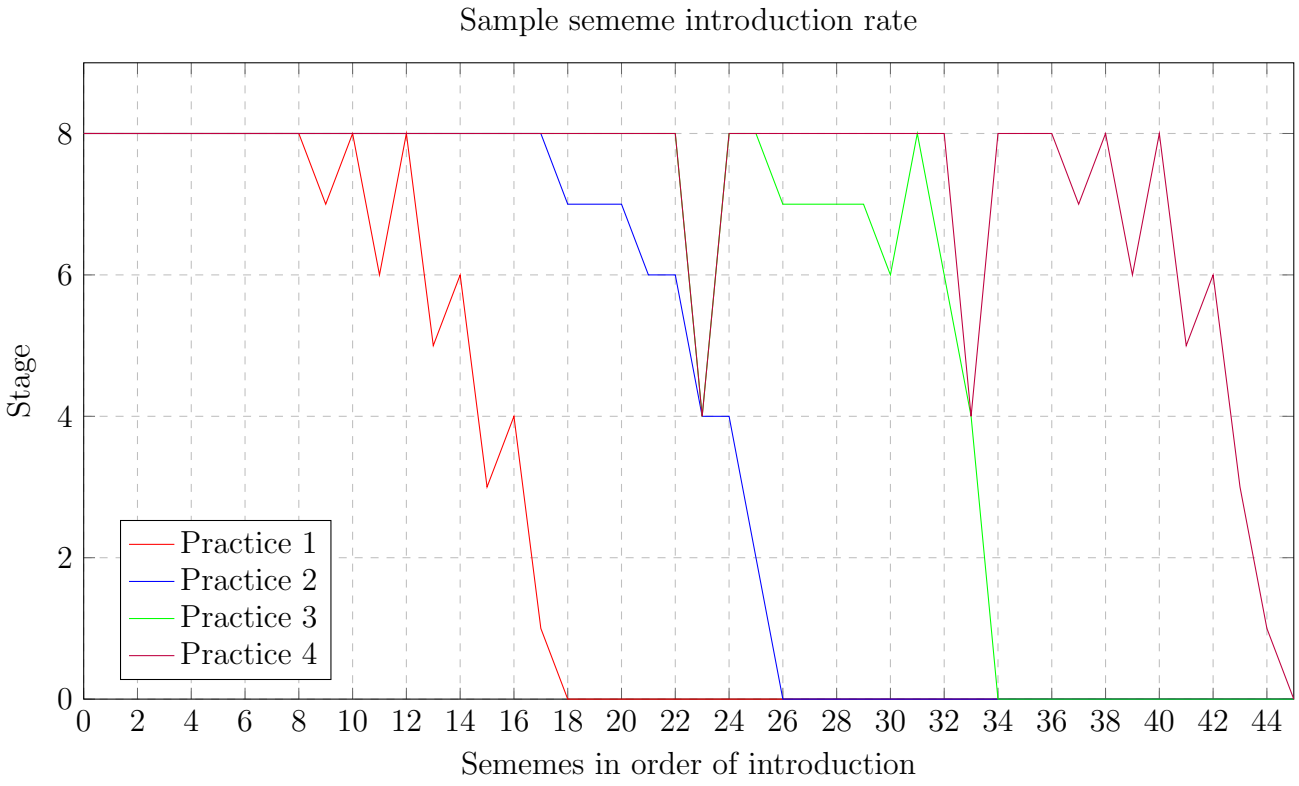

Figure 5.1 The stages of every introduced sememe after taking one, two, three, and four five-minute practices, in the order that the sememes were introduced.

position at which new sememes are introduced.

The sememes in the plot are listed in the exact order that they were introduced, i.e. sememe 0 is "to be", sememe 1 is "to eat", following the order outlined in Appendix B.3. After the four practices, 45 sememes had been introduced. As the graph shows, after each consecutive lesson, the sememes that were new in the previous one have moved up to stage 8 to make way for new ones. No sememes are missed and each one is properly introduced. Occasionally, sememes make it to a higher stage than some sememes introduced before them, but that is to be expected, especially since there is less 'competition' for certain categories. For example, there are only three sememes in the category `food_descriptor`, but there are 15 in the category `food_item`, meaning that if a random sememe for either category is required, it is more likely for a certain food descriptor to be chosen than for a certain food item, causing the food descriptors to level up faster.

At the end of each practice, there were always either 7 or 8 sememes in 'active rotation', or

with a stage that is not 0 or 8. From this it can be concluded that there are never more than 7-8 different sememes that are new to a user at a time, which strikes a neat balance between giving variation in exercises and not being overwhelming.

When disregarding outliers, each graph has roughly the same shape, suggesting that sememes get introduced at a similar pace as the course goes on. The sole reason for concern would be sememes 23 ("chicken") and 33 ("wine"), which seem to get stuck on stage 4. To remedy this, the queuing algorithm would probably benefit from an occasional quick scan ahead, to check whether there are sememes that get left behind, and pull them forward.

5.3 Android testing

5.3.1 Database unit testing

Before putting all components together, one more should be tested separately, which is the Room database. Android development allows for testing databases separate from the application interface through standard Java unit tests. As Kotlin is fully interoperable with Java, these tests work with Kotlin as well.

A test was written which constructs the database in the same way as it is constructed during the app's initialization, with the exception that for the purposes of the test, database calls on the main thread are allowed, since there is no other work to be done. The test than makes a call to the database for every query that is specified in the DAO (see Section 4.2.3), including inserting entries, updating entries, retrieving entries by ID and `learned` status, and clearing tables. Each query was verified to work correctly.

5.3.2 Application experience

The final element to test is the user experience. While the project could not be tested on human subjects, it was still deemed important to ensure that 1) performance is fast, 2) input is fluid, with no freezes, 3) the application respects the user's privacy, and 4) the *Language Lion* respects

Stockwell and Hubbard's principles of mobile language learning as outlined in Section 2.1.

The application was tested on a Google Pixel 2 XL, a flagship-level Android device from late 2017, running the latest version of Android, Android 10. There are currently no known bugs that lead to crashes. Performance throughout the app was found to be good, with no discernible waiting time or lag between screens, with the exception of the start of a lesson. As has already been described in Section 4.2.5, since the data files can only be accessed by copying them to the application cache, this has a measurable performance impact. As this freezes the UI, to account for this, a loading spinner was implemented when a lesson is initialized to provide visual feedback to the user.

The *Language Lion* does not require any device permissions such as access to the camera, microphone, or files. It also will never connect to the internet and does not save any personal details. While it is possible to answer the exercises using voice input, by pressing the microphone button on the software keyboard, all voice input is handled through the user's keyboard application and the *Language Lion* never receives microphone access directly.

Most of Stockwell and Hubbard's principles have been followed. Environmental distractions are kept to a minimum by providing a clean, simple user interface during a lesson. The *Language Lion* is designed to be inclusive by using clear and large typography, an accent color that is standardized in Android and thereby verified to provide sufficient contrast, and clear buttons with wide spacing between them. In addition, it does not require any internet connection at all. Task goals are always specified at the top of the lesson screen, and users have control over the duration of their own practice sessions. Only two pieces of advice are not implemented: the *Language Lion* does not feature any push notifications, to motivate learners to come back, nor does it feature any tools for teachers. Both features should be implemented in future versions of the app.

Chapter 6

Discussion

There is an application—it can be installed, per the instructions in Appendix A, and it will work, teaching anyone how to say "I am a banana" in Dutch with complete and utter confidence. It would seem that automated exercise generation in mobile language learning is possible.

The actual question, of course, is how well it functions. This project was not an experiment designed to prove or reject a hypothesis; instead, it was an exploration, identifying a potential next step for mobile language learning (Chapter 2), describing a vision for that step (Chapter 3), and implementing a basic realization of that vision (Chapter 4) in order to identify any roadblocks (Chapter 5) and formulate advice on how to proceed (right here).

The project's clearest contribution to the field has been the creation and evaluation of the Sememe as a useful concept in mobile language learning. While differences in languages and cultures are too big for each concept to be described as a sememe that maps translations of lexemes, it has proven a solid base for basic language learning, especially if the languages, like English and Dutch, are closely related. Coupled with a context-free grammar, even a few sememes can generate thousands of sentences, and while some of those will always be more relevant in the real world than others, each sentence provides its own opportunity to practice and think about language.

Compared to the original vision, many features landed on the cutting room floor: precise

control over practice time, a categorized notebook with personalized Starred and Difficult sections, dictation exercises, grammar tips, and most notably an advanced spaced repetition algorithm that reintroduces sememes based on the student's performance, rather than a set schedule. All of these features already existed, however, in existing applications, in one way or another. What did not exist was a translation exercise generation algorithm. The *Language Lion* generates sentences with a subject, verb, optional complement, and optional interjection, with support for all persons of the present simple and all different combinations of determiners. It is thereby more sophisticated than any similar project to date.

It has also clearly uncovered the shortcomings of its approach. The biggest flaw in the current implementation is the absence of support for continuous verb forms, which are grammatically correct translations of many of the generated sentences. One of the main areas for future research could therefore be to build a map of Dutch and English sentence structures, similar to the map of lexemes that the Semanticon provides, instead of the current one-template-fits-all approach of the Grammar. This would probably need to be paired with upgrades to the SimpleNLG engine that currently only allows for generating one set word order at a time.

One area that this project has not looked into at all is detailed error feedback. In fact, when a question in answered incorrectly, a new exercise with the main sememe is created, regardless of whether that was the sememe that the error was made in. Subsequent versions of the application should implement an actual answer processor that analyzes the answer and gives the user specific feedback.

After that, translation exercises are only the tip of the iceberg of the potential of NLG in language learning. More exercise types and more personalization have already been mentioned, but there are also the other phases of natural language generation: content determination and sentence planning (see Section 2.3). Instead of generating exercises that stand independent of each other, NLG might be used to procedurally generate an entire conversation, such as an interaction in a restaurant or hotel for the student to take a role in, or a short story, that the student has to answer questions about. One novel idea certain to ruffle some feathers is—

A green owl appears in the corner of my screen. Alas, it looks like I need to go and practice.

APPENDICES

Appendix A

Installing the application

The *Language Lion* is not published on the Google Play Store, but can be downloaded from the Releases tab of the project's GitHub page here. Select the `language-lion.apk` file under "Assets" and download it to your device.

To install it on an Android phone, the phone needs to have the "install unknown apps" setting enabled. As many Android manufacturers make their own changes to the Settings app, the location of this setting will vary per device—a Google search for "install APK files on Android" should pull up a variety of guides that walk through the entire process for each phone. When this setting is enabled, simply copy the downloaded APK to your phone (most conveniently to the Downloads folder) and tap on it there. Remember that the app will only run on devices with Android 6.0 or higher. Ignore any warnings that Google cannot verify the application's developer and let the app install anyway. The installation process should not take longer than a minute, and when it is done, the app will show up in your app drawer, between all other apps.

Enjoy your first Dutch lessons!

Appendix B

Data sets

All that is taught in the course.

B.1 Complete Semanticon

Table B.1 The complete Semanticon.

ID	Name	English	Dutch	Categories
S0010000	greeting_quick	hi, hey	hoi	interjection
S0010001	greeting	hello	hallo	interjection
S0010002	bye_quick	bye	dag	interjection
S0010003	bye_informal	see you, see ya	doei, tot ziens	interjection
S0010004	cheers	cheers	proost	interjection
S0010005	goodmorning	good morning	goedemorgen	interjection
S0010006	goodafternoon	good afternoon	goedemiddag	interjection
S0010007	goodevening	good evening	goedenavond	interjection
S0010008	goodnight	good night	goedenacht	interjection
S0010009	yes	yes	ja	interjection
S0010010	no	no	nee	interjection
S0010011	please	please	alsjeblieft, alstublieft	interjection
S0010012	thanks	thanks, thank you	dank je, dank u, dank je wel, dank u wel	interjection
S0010013	sorry	I'm sorry, sorry	het spijt me, sorry	interjection
S0010014	excuseme	excuse me, pardon me	pardon	interjection
S0010015	okay	okay, ok	oké, ok	interjection
S0020000	the	the	de, het	-

ID	Name	English	Dutch	Categories
S0020001	a	a	een	-
S0021000	i	I	ik	person
S0021001	me	me	mij, me	pers_pron_obj
S0021002	you_sub	you	jij, je, u, jullie	person
S0021003	you_obj	you	jou, u, jullie	pers_pron_obj
S0021004	he	he	hij	person
S0021005	him	him	hem	pers_pron_obj
S0021006	she	she	zij, ze	person
S0021007	her	her	haar	pers_pron_obj
S0021008	it	it	het	person pers_pron_obj
S0021009	we	we	wij, we	person
S0021010	us	us	ons	pers_pron_obj
S0021012	they	they	zij, ze	person
S0021013	them	them	hen	pers_pron_obj
S0030000	be	be	zijn	verb_intransitive verb_transitive verb_copular verb_food verb_drink
S0030001	have	have	hebben	verb_intransitive verb_food verb_drink
S0031001	eat	eat	eten	verb_intransitive verb_food
S0031002	drink	drink	drinken	verb_intransitive verb_drink
S0031003	speak	speak	spreken	verb_intransitive verb_language
S0031004	go	go	gaan	verb_intransitive
S0031005	walk	walk	lopen	verb_intransitive
S0031006	run	run	rennen	verb_intransitive
S0031007	cycle	cycle, bike	fietsen	verb_intransitive
S0031008	help	help	helpen	verb_intransitive verb_transitive
S0031009	read	read	lezen	verb_intransitive verb_transitive verb_readable verb_language
S0031010	live_in	live	wonen	verb_intransitive verb_location
S0031011	live_alive	live	leven	verb_intransitive

ID	Name	English	Dutch	Categories
S0031012	see	see	zien	verb_transitive
S0031013	hear	hear	horen	verb_transitive
S0040000	man	man	man	person
S0040001	woman	woman	vrouw	person
S0040002	boy	boy	jongen	person
S0040003	girl	girl	meisje	person
S0040004	child	child, kid	kind	person
S0041000	apple	apple	appel	food_item
S0041001	banana	banana	banaan	food_item
S0041002	strawberry	strawberry	aardbei	food_item
S0041003	bread	bread	brood	food_item no_determiner
S0041005	cheese	cheese	kaas	food_item no_determiner
S0041006	soup	soup	soep	food_item no_determiner
S0041007	pasta	pasta	pasta	food_item no_determiner
S0041008	breakfast	breakfast	ontbijt	food_item no_determiner
S0041009	lunch	lunch	lunch	food_item no_determiner
S0041010	dinner	dinner	avondeten	food_item no_determiner
S0041011	chicken	chicken	kip	food_item
S0041012	fruit	fruit	fruit	food_item no_determiner
S0041013	vegetable	vegetable	groente	food_item
S0041014	egg	egg	ei	food_item
S0041015	water	water	water	drink_item no_determiner
S0041016	milk	milk	melk	drink_item no_determiner
S0041017	juice	juice	sap	drink_item no_determiner
S0041018	coffee	coffee	koffie	drink_item no_determiner
S0041019	tea	tea	thee	drink_item no_determiner
S0041020	beer	beer	bier	drink_item no_determiner

ID	Name	English	Dutch	Categories
S0041021	wine	wine	wijn	drink_item no_determiner
S0042000	menu	menu	menu, kaart	readable
S0042001	book	book	boek	readable
S0042002	newspaper	newspaper	krant	readable
S0050000	good	good	goed	people_descriptor food_descriptor drink_descriptor
S0050001	bad	bad	slecht	people_descriptor food_descriptor drink_descriptor
S0050002	tasty	tasty	lekker	people_descriptor food_descriptor drink_descriptor
S0051000	dutch	Dutch	Nederlands	locale
S0051001	english	English	Engels	locale

Note a few absent sememes. S0020011 was the original sememe for the "plural you" or "jullie", but was taken out after the consolidation as described in Section 4.1.2. S0031000, "be_called", was originally going to be used for teaching the student how to introduce themselves, but was taken out as neither the passive voice nor proper nouns are supported. S0041004, "slice of bread", was taken out as the English term here consists of multiple words, which is also not supported.

B.2 Complete Grammar

Table B.2 The complete Grammar.

ID	Subject	Verb	Complement
G001	person	verb_intransitive	-
G002	person	verb_food	food_item
G003	person	verb_drink	drink_item
G004	food_item	verb_copular	food_descriptor
G005	drink_item	verb_copular	drink_descriptor
G006	person	verb_readable	readable
G007	person	verb_copular	people_descriptor
G008	person	verb_copular	locale
G009	person	verb_language	locale
G010	person	verb_transitive	pers_pron_obj

B.3 Initial queue order

This is the order in which the sememes are introduced. The sememes "the", "a", and "I" are automatically unlocked from the start.

1. be
2. eat
3. drink
4. apple
5. banana
6. water
7. you_sub
8. strawberry
9. bread
10. good
11. cheese
12. bad
13. soup
14. he
15. she
16. pasta
17. breakfast
18. milk
19. lunch
20. juice
21. dinner
22. speak
23. chicken
24. coffee

25. we
26. fruit
27. tea
28. they
29. vegetable
30. beer
31. tasty
32. egg
33. wine
34. have
35. Dutch
36. English
37. go
38. man
39. walk
40. woman
41. run
42. boy
43. cycle
44. girl
45. help
46. child
47. see
48. me
49. you_obj
50. hear
51. him
52. greeting_quick
53. her
54. greeting
55. it
56. bye_quick
57. us
58. bye_informal
59. them
60. cheers
61. goodmorning
62. goodafternoon
63. goodevening
64. goodnight
65. read
66. yes
67. no
68. menu
69. please
70. thanks
71. book
72. sorry
73. excuseme
74. newspaper
75. okay

Bibliography

Amaral, L., Meurers, D., & Ziai, R. (2011). Analyzing learner language: towards a flexible natural language processing architecture for intelligent language tutors. *Computer Assisted Language Learning*, *24*(1), 1–16.

Atkinson, R. C. (1972). Optimizing the learning of a second language vocabulary. *Journal of Experimental Psychology*, *96*(1), 124–129.

Baddeley, A. D. & Longman, D. J. A. (1978). The influence of length and frequency of training session on the rate of learning to type. *Ergonomics*, *21*(8), 627–635.

Beatty, K. (2010). *Teaching and researching computer-assisted language learning* (2nd ed.). Pearson Education Ltd.

Blom, A. (1997). Nederlands leren: regels of voorbeelden? *Levende Talen*, *524*, 582–585.

Bloom, K. C. & Shuell, T. J. (1981). Effects of massed and distributed practice on the learning and retention of second language vocabulary. *The Journal of Educational Research*, *74*(4), 245–248.

Boom Uitgevers. (2020). Boom NT2. Retrieved from https://www.nt2.nl/nl

Burstein, J. & Marcu, D. (2005). Translation exercise assistant: automated generation of translation exercises for native-Arabic speakers learning English. In *Proceedings of hlt/emnlp on interactive demonstrations* (pp. 16–17). Association for Computational Linguistics.

Busuu Ltd. (2020). Busuu. Retrieved from https://www.busuu.com/

Cepeda, N. J., Pashler, H., Vul, E., Wixted, J., & Rohrer, D. (2006). Distributed practice in verbal recall tasks: a review and quantitative synthesis. *Psychological Bulletin*, *132*(3).

Chinnery, G. M. (2004). Going to the MALL: mobile assisted language learning. *Language Learning and Technology*, *10*(1), 9–16.

Cleron, M. (2017, May 17). Android announces support for Kotlin. Retrieved from https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html

Council of Europe. (2011). Common European framework of reference for languages: learning, teaching, assessment. Retrieved from https://www.coe.int/en/web/language-policy/home?e1_en.asp

De Jong, R. & Theune, M. (2018). Going Dutch: creating SimpleNLG-NL. In *Proceedings of the 11th international natural language generation conference*, November 5–8, 2018 (pp. 73–78). Tilburg, the Netherlands. Association for Computational Linguistics.

Degand, L. (1993). Towards a systemic functional grammar of Dutch for multilingual text generation. In *Proceedings of the 4th European workshop on natural language generation*, April 28–30, 1993 (pp. 143–147). Pisa, Italy. Association for Computational Linguistics.

Drops. (2020). Drops. Retrieved from https://languagedrops.com/

Duolingo. (2020). Duolingo. Retrieved from https://www.duolingo.com/

Gatt, A. & Reiter, E. (2009). SimpleNLG: a realisation engine for practical applications. In *Proceedings of the 12th European workshop on natural language generation*, March 30–31, 2009 (pp. 90–93). Athens, Greece. Association for Computational Linguistics.

Gilbert, N. & Keet, C. M. (2018). Automating question generation and marking of language learning exercises for isizulu. In *Cnl* (pp. 31–40).

Google Inc. (2020a). Bottom navigation. Retrieved from https://material.io/components/bottom-navigation

Google Inc. (2020b). The color system. Retrieved from https://material.io/design/color/the-color-system

Haase, C. (2019, May 7). Google I/O 2019: empowering developers to build the best experiences on Android + Play. Retrieved from https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html

Heift, T. (2001). Intelligent language tutoring systems for grammar practice. *Zeitschrift für Interkulturellen Fremdsprachenunterricht*, *6*(1).

Krajina, T. (2020). 10,000 sentences. Retrieved from https://github.com/tkrajina/10000sentences

Kuiken, F. (2017). Nederands leren: makkelijk of moeilijk? *Les*, *35*(204), 22–25.

Lesson Nine GmbH. (2020). Babbel. Retrieved from https://www.babbel.com/

Lingvist. (2020). Lingvist. Retrieved from https://lingvist.com/

Malafeev, A. (2014). Automatic generation of text-based open cloze exercises. In *International conference on analysis of images, social networks and texts* (pp. 140–151). Springer.

Marsi, E. (1998). A reusable syntactic generator for Dutch. In *In* (pp. 171–194). Rodopi.

Memrise. (2020). Memrise. Retrieved from https://www.memrise.com/

Ministerie van Sociale Zaken en Werkgelegenheid. (2015). Naar Nederland. Retrieved from https://www.naarnederland.nl/

Pimsleur, P. (1967). A memory schedule. *The Modern Language Journal*, *51*(2), 73–75.

Reiter, E. & Dale, R. (1997). Building natural language generation systems. *Natural Language Engineering*, *3*(1), 57–87.

Rosetta Stone Ltd. (2020). Rosetta Stone. Retrieved from https://www.rosettastone.com/

Russell, S. J. & Norvig, P. (2010). *Artificial intelligence: a modern approach* (3rd ed.). Pearson Education Ltd.

Schepens, J. J. (2015). *Bridging linguistic gaps: the effects of linguistic distance on the adult learnability of Dutch as an additional language*. Utrecht: LOT.

Settles, B. & Meeder, B. (2016). A trainable spaced repetition model for language learning. In *Proceedings of the 54th annual meeting of the association for computational linguistics*, August 7–12, 2016 (pp. 1848–1858). Berlin, Germany. Association for Computational Linguistics.

Steel, C. H. (2012). Fitting learning into life: language students' perspectives on benefits of using mobile apps. In M. Brown, M. Hartnett, & T. Stewart (Eds.), *Future challenges, sustainable futures* (pp. 875–880). Wellington: Proceedings ascilite.

Stockwell, G. & Hubbard, P. (2013). Some emerging principles for mobile-assisted language learning. Retrieved from http://www.tirfonline.org/english-in-the-workforce/mobile-assisted-language-learning/

Van der Lee, C., Krahmer, E., & Wubben, S. (2017). PASS: a Dutch data-to-text system for soccer, targeted towards specific audiences. In *Proceedings of the 10th international conference on natural language generation*, September 4–7, 2017 (pp. 95–104). Santiago de Compostela, Spain. Association for Computational Linguistics.

Vesselinov, R. (2009). Measuring the effectiveness of Rosetta Stone. Retrieved from http://resources.rosettastone.com/CDN/us/pdfs/Measuring_the_Effectiveness_RS-5.pdf

Vesselinov, R. & Grego, J. (2012). Duolingo effectiveness study. Retrieved from http://static.duolingo.com/s3/DuolingoReport_Final.pdf

Vesselinov, R. & Grego, J. (2016). The Babbel efficacy study. Retrieved from https://press.babbel.com/en/releases/downloads/Babbel-Efficacy-Study.pdf