Automation of Formative Assessment: Implementation and Evaluation of an Artificial Teaching Assistant

JOHAN MYRSMEDEN
Automation of Formative Assessment: Implementation and Evaluation of an Artificial Teaching Assistant

JOHAN MYRSMEDEN

Master of Science in Engineering and in Education
Date: June 18, 2018
Supervisor: David Broman, Department of Software and Computer System, KTH
Co-supervisor: Fredrik Enoksson, Department of Learning, ITM, KTH
Examiner: Stefan Stenbom, Department of Learning, ITM, KTH
Swedish title: Automatisering av formativ bedömning: implementering och utvärdering av en artificiell lärarassistent
School of Industrial Engineering & Management
Abstract

The Swedish government has decided to add programming to the Swedish curriculum to strengthen students’ digital skills. The teachers who will teach programming do not always know programming themselves. Because of that, KTH Royal Institute of Technology researchers are planning to start an initiative of creating a Massive Open Online Course (MOOC) in fundamental programming in Swedish for those teachers.

Interpreting error messages given by the compiler is one problem with learning programming. An aim of this study is to lower that threshold. The study seeks to identify common misconceptions about programming among novice programmers in order to design a static analyser that investigates code written by students and provides formative feedback to help students in their own learning process.

The study combines the constructivist theory of learning with views about formative assessment in order to automate the process that is usually done by a teacher or teaching assistant when assessing code. A phenomenographic study is done in order to identify teachers’ perceptions of common misconceptions about programming by interviewing five active teachers at KTH. The results of that study are used to construct code examples that correspond to these misconceptions. Those results lead to the design and implementation of a software that detects these problems in code. That software is evaluated using a larger set of test data, consisting of 77 errors divided into five larger programs, inserted by independent individuals.

From the initial study, five categories of misconceptions are given. Of the 77 errors, the majority are correctly positioned and almost all are given a good hint about the position. About a quarter of the errors are parse errors, which never reach the analysing part of the software that demands the program to be parsable. The study shows that we have succeeded both in designing and implementing a software that detects the identified misconceptions with good results. In the context of a MOOC, the software might require an extension with a more advanced parser and also dynamic analysis to be able to test the correctness of the students’ programs. The software is limited to handle the language Javascriptish, which is a subset of JavaScript.

Keywords: constructivism, formative assessment, automated assessment, computer-aided education, computer-aided assessment
Sammanfattning

För att stärka den digitala kompetensen bland svenska grundskole- och gymnasieelever har programmering lagts in i läroplanen. De lärare som är tänkta att hålla i denna programmeringsundervisning har själva inte alltid tillräckliga kunskaper i programmering, varpå ett initiativ för att hålla en storskalig, öppen och internetbaserad kurs (MOOC) på svenska har startats av forskare på Kungliga Tekniska Högskolan (KTH).

En av svårigheterna med att lära sig programmering är att förstå meddelanden som kompilatorn ger. Ett mål med denna studie är att sänka denna tröskel för studenten. Denna studie ämnar identifiera vanliga missuppfattningar om programmering hos nybörjare, för att designa en programvara som utför statisk analys av kod skriven av studenter och därefter ge återkoppling kring dessa missuppfattningar.

Studien kombinerar den konstruktivistiska teorin om lärande med tankar om formativ bedömning för att automatisera den process som vanligtvis görs av en lärare eller lärarassistent vid bedömning av kod. En fenomenografisk studie görs för att identifiera lärarens uppfattningar kring programmering genom att intervja fem verksamma lärare på KTH. Resultatet från den studien används sedan för att konstruera kodexempel som belyser de identifierade missuppfattningarna. Därefter designas och implementeras en mjukvara som sedan evalueras genom att analysera totalt 77 fel, konstruerade av oberoende individer, uppdela på fem större program.

Den initiala studien resulterar i fem olika kategorier av missuppfattningar. Av de 77 evaluerade felen ger majoriteten en korrekt positionsangivelse och nästan alla ger en god indikation över var felet ligger. Omkring en fjärdedel av felen är parsningsfel, vilka aldrig när huvudmjukvaran som kräver att programmet är parsningsbart.

Studien visar att vi lyckas designa och implementera en programvara som med goda resultat upptäcker vanliga missuppfattningar kring programmering hos nybörjare, baserad på det teoretiska ramverket. I kontexten av en kurs på internet kan programvaran behöva utvecklas med en mer avancerad syntaxanalys (eng. parser) samt lägga till dynamisk analys av program för att även kunna testa programmens korrekthet. Programvaran är begränsad till att analysera kod skriven i språket JavaScriptish, vilken är en delmängd till JavaScript.

Nyckelord: konstruktivism, formativ bedömning, automatiserad bedömning, datorstödd undervisning, datorstödd bedömning
# Contents

1 Introduction 1  
1.1 Background 1  
1.2 Purpose and aim 3  
1.3 Research question and problems 3  
1.4 Delimitations 4  
1.5 Context and contributions 4  
1.6 Outline of the thesis 5  

2 Related work 7  
2.1 Introductory courses 7  
2.2 Misconceptions about programming 8  
2.3 Formative Assessment 9  
2.4 Automated assessment 10  
2.5 Ontology and epistemology in Computer Science Education 12  
2.6 Compiler Error Messages and Learning 16  

3 Methodology 19  
3.1 Development of a theory about common misconceptions among novice programmers 19  
3.2 Development of the Artificial Teaching Assistant 21  
3.3 Evaluation of the Artificial Teaching Assistant 22  
3.4 Ethical aspects, societal relevance and sustainability 23  

4 Results from the Investigation of Common Misconceptions 25  
4.1 Interview A 25  
4.2 Interview B 26  
4.3 Interview C 27
CONTENTS

4.4 Interview D ........................................... 28
4.5 Interview E ........................................... 29
4.6 Analysis ............................................. 30

5 Design and Implementation ........................................... 33
  5.1 Design of the ATA ....................................... 33
    5.1.1 Requirements ....................................... 36
  5.2 Implementation of the ATA ................................... 36
    5.2.1 Defining Javascriptish ................................. 37
    5.2.2 Characteristics of the misconceptions ................. 37
    5.2.3 Detecting misconceptions .............................. 40
    5.2.4 Reporting misconceptions .............................. 43

6 Evaluation, Results and Discussion .............................. 47
  6.1 Evaluation ............................................ 47
    6.1.1 Results ............................................ 51
  6.2 Discussion ............................................ 54

7 Conclusion .................................................. 65
  7.1 Future research ......................................... 65
  7.2 Conclusion ............................................. 66

Bibliography ..................................................... 67

A Evaluation Programs ............................................. 71
  A.1 Binary search .......................................... 71
  A.2 Bubble sort ............................................ 73
  A.3 Find the smallest number divisible by a range ............ 74
  A.4 List primes ............................................. 75
  A.5 Polynomial division ..................................... 76

B Source code of the ATA ........................................... 81
Chapter 1

Introduction

This thesis is targeting the intersection of the constructivist theory of learning, automation, and formative assessment. The aim is to automate the interaction between students and teachers to provide feedback in the context of assignments in introductory programming. This first chapter contains:

- The background of the research area.
- Purpose and aim of the thesis.
- Research questions and problems.
- Delimitations of the thesis.
- The context of the thesis, and contributions.
- An outline for the remainder of the thesis.

1.1 Background

As a step towards strengthening the digital skills among Swedish elementary and upper secondary school students, elements of programming, among other digitalisation centred competences, have been added to the Swedish curriculum (Regeringskansliet, 2017). However, many teachers that are supposed to teach programming in Fall 2018 lack a proper education in teaching programming (Thurfjell, 2017). Hence, many teachers have to be educated in a short amount of time, but there
are certain restrictions that make an ordinary university course unsuitable. Examples of such factors include the fact that one cannot fit too many students in one lecture hall. This means that courses that can be scaled up to manage many participants have to be designed, which makes a Massive Open Online Course, MOOC, suitable for this case.

A MOOC is an alternative solution to the regular educational model. According to HOME (2015), these courses are open for everyone and designed for a large amount of participants. Thus, they are designed to easily be scaled for many users. The learning activities are carried out online via for instance recorded lectures and interactive exercises. Since a MOOC is online, the possibilities of feedback through physical interaction with teachers and students are reduced. MOOCs often need some kind of system where the interaction between course participants and teacher can happen, which is usually managed in so called Learning Management Systems.

Previous studies have investigated how students use the discussion forums in a Learning Management System, which is one possible way for interaction and feedback. A study by Zhu et al. (2015) concluded that discussion forums as they are designed today may not be the best tool for programming education since the forums are text-based and there is a need for a more visual communication method to discuss concepts such as mental models of the computer memory. Since the Learning Management Systems might not be adapted to programming courses and with a growing need for MOOCs in programming, one might draw the conclusion that there is a need for other types of feedback tools in programming.

This thesis works in the intersection between the constructivist theory of learning, formative assessment and automation where the Artificial Teaching Assistant (ATA) is constructed (see Figure 1.1) by combining parts of the constructivist theory of knowledge and formative assessment in order to automate that process. According to the constructivist theory of knowledge, the student constructs knowledge based on their current knowledge and hence less usable models may be built and misconceptions will occur. A task for the teachers is to identify these misconceptions among their students, try to understand why they occurred and provide an explanation that helps the student build a more viable model – which is one part of formative assessment.
1.2 Purpose and aim

The purpose with this thesis is to automate the part of formative assessment that focuses on gaps in the students’ cognitive models and the feedback required to fill in those gaps, by building an *Artificial Teaching Assistant* (ATA). The ATA is a software that can analyse code written by students and provide feedback that helps the student improve their programming skills.

1.3 Research question and problems

The aim of this thesis is to automate formative assessment in the context of introductory courses in programming. Hence, the main focus is to analyse the code written by the students and provide feedback about possible misconceptions that the student may have. This leads to the following research question and problem:

- Which are the common misconceptions in programming for novice programming students?
- Problem: To design a static analyser that detects common misconceptions about programming and provides feedback about what, why and where.
A static analyser is a software that performs analysis of a computer software without executing it. The code written by the students will be converted into a tree structure, which is traversed by the software detecting certain patterns that indicate the misconceptions.

The problem is to lower the learning threshold given by the compilers of today. The evaluation of the problem consists of two parts: see whether an error is detected or not, and if the detected position of the error is correct.

To concretise the formative assessment, it is divided into the three keywords what, why and where in the problem formulation. What and why are two qualitative measurements, that focus on the cause of the problem and its explanation. Where focuses on the position of the error and is a quantitative measurement.

1.4 Delimitations

The design and implementation of the ATA are not generalised when it comes to the choice of programming language. Instead, the analysed code should be written in the language Javascriptish, which is a subset of JavaScript and is explained in more detail in Chapter 5.

Furthermore, the chosen programming language focuses on the imperative programming paradigm. This means that the functional or object-oriented programming paradigm will not be taken into consideration during the design and implementation of the ATA.

Formative assessment is a broad concept, and in order to narrow it down for the scope of this thesis, the holistic perspective of the learning process of the student is not considered in the implementation. The holistic perspective is discussed in Chapter 6.

1.5 Context and contributions

This thesis is a part of a larger project where the aim is to create a MOOC in fundamental programming, which targets teachers that should learn programming. Also, the MOOC can be used in undergraduate courses at KTH, or as a preparation course.

There are many things that should be in place in order to succeed with a MOOC. This thesis only focuses on the part that analyses the
code the students are writing when they are doing exercises on their own.

The supervisor of this thesis, David Broman, is responsible for the MOOC and has also written the parser and lexer that delivers the abstract syntax tree for the ATA to analyse.

This thesis aims to use static analysis in order to find the common misconceptions which will be the foundation for enhanced error messages. Previous studies have tried to identify students’ difficulties and possible areas of misconceptions in introductory programming courses, specifically where Java and object-oriented programming are used (Du Boulay, 1986; Altadmri and Brown, 2015). This study aims to investigate common misconceptions in more fundamental programming where the object-oriented paradigm is not present and focus on the strict imperative features such as variables and functions.

Studies have been done to investigate how students interpret error messages, and the effect of enhanced compiler error messages (Prather et al., 2017). Truong, Roe, and Bancroft (2004) have used static analysis to compare students’ solutions to a model solution. However, the ATA should not require the existence of a model solution in order to find the common misconceptions.

Finally, many of the previous studies are lacking a theoretical foundation when it comes to education. Therefore, this thesis will add dimensions of formative assessment to build the error messages and use the constructivist theory of knowledge as a foundation for the design, implementation, evaluation and analysis of the ATA. These theories are explained in more detail in Chapter 2.

1.6 Outline of the thesis

The thesis is organised as follows:

- Chapter 2 presents previous work related to this thesis.
- Chapter 3 contains the methodology description, divided into two parts: one concerning the investigation of common misconceptions and one for the development of the ATA.
- Chapter 4 presents the results and analysis of the investigation of common misconceptions in programming among novice programming students.
Chapter 5 describes the design and implementation of the ATA with respect to the previous analysis.

Chapter 6 presents and discusses the results of the evaluation of the ATA.

Chapter 7 contains suggestions for future research and concludes the thesis.
Chapter 2

Related work

This chapter contains a description of previous work related to this thesis. The chapter covers:

- Introductory courses in programming.
- Some notes about misconceptions about programming.
- A description of formative and summative assessment.
- An introduction to automated assessment.
- A description of the constructivist perspective of learning, related to computer science.
- Some previous studies about Compiler Error Messages.

2.1 Introductory courses

Pears et al. (2007) discuss the subject “learning programming” and mentions that one view is that programming is mathematically based, which means that in order to be a good programmer one has to be able to prove the correctness of their program (cf. Denning (1989) and Hoare (1969)). Robins, Rountree, and Rountree (2003) conclude that most textbook writers focus on syntax and language features.

Du Boulay (1986) has identified different areas of difficulties that novice programmers encounter when they start their journey towards becoming programmers.
These are:

1. **orientation**; finding out what programming is for and which types of problems that can be solved with the help of programming

2. **the notional machine**; learning the behaviour of the computer

3. **notation**; the syntax and semantics of the particular programming language that is being learned

4. **structures**; different schemas or plans that are being used when you program, for instance how one can compute a sum using a loop

5. **pragmatics**; the whole programming process such as specification, development, testing and debugging.

Something indicated by the study conducted by Pears et al. (2007) is that students want to learn languages that they can use after graduation. This is something that talks against languages constructed specifically for education such as LOGO\(^1\) or Eiffel\(^2\) (cf. Pears et al. (2007) and Meyer (2003)).

Another problem with teaching introductory programming, or programming at all, is that the teacher is using a self-centred perspective and is using her or his own way of thinking about the computer and programming as a starting point and not the students perspective. Furthermore, student problems can become ignored by teachers that reflect on their own learning process as a way to relate to student difficulties (Berglund et al., 2009; Berglund and Lister, 2010).

### 2.2 Misconceptions about programming

Du Boulay (1986) has investigated common mistakes done by novice programmers due to misapplications of analogies. For instance, one common mistake is to compare a variable with a box, which makes some students think that a variable can hold more than a single value since a box can contain multiple objects. This is also supported by Ben-Ari (2001).  

\(^1\)http://el.media.mit.edu/logo-foundation/index.html  
\(^2\)https://www.eiffel.org/
One common problem according to Du Boulay (1986) derives from the asymmetry in the assignment operator, that is the sign $=$ for instance. A common exercise is to interchange the values of two variables, which requires the use of a third temporary variable.

```
temp = a
a = b
b = temp
```

Since many students have not understood the sequential nature of these statements, they get the assignments in the wrong order and hence get an incorrect answer. As a consequence of this, beginners also are puzzled by an assignment as

```
a = a + 1
```

When it comes to syntax, many syntactic errors are associated with the lack of the semicolon.

### 2.3 Formative Assessment

Assessment may be divided into *formative assessment* and *summative assessment*, each of which has a different purpose. Summative assessment summarises your achievements and gives you a grade and is something you often get by the end of a course. Formative assessment aims to give feedback to the student with the objective to both see what the student knows at the moment as well as giving feedback on how they can proceed. According to Black and Wiliam (2009), formative assessment should not be equipped with a grade since it is desirable to prevent giving a mark to help the student focus on the feedback rather than the mark itself. Ramaprasad (1983) argues that feedback is not the information about a certain gap of knowledge, it is only feedback if the information can be used to alter the gap. As a consequence, the teacher should not tell the student where the error is and nothing more, instead they should provide sufficient information to let the student figure out what went wrong and why to help them correct their code.

According to Black and Wiliam (2009, p.7), formative assessment is about three key processes:

- Establishing where the learners are in their learning.
- Establishing where they are going.
• Establishing what needs to be done to get them there.

and also five main types of activities Black and Wiliam (2009, p.7):

• Sharing success criteria with learners.

• Classroom questioning.

• Comment-only marking.

• Peer- and self-assessment.

• Formative use of summative tests.

Formative assessment is the whole philosophy of using assessment to improve education. It is not only about the techniques and activities teachers use. Wiliam and Leahy (2015) point out that when designing learning activities, one should start with a clear view of the learning that is intended, and then design the activities: “by being clear about what it is we want students to learn [...] it is more likely that our students will learn what we need them to learn” (p. 27). Assessment is formative when it is used to modify teaching and learning activities (Wiliam, 2016).

Wiliam and Leahy (2015) talk about two kinds of feedback: corrective and reinforcing. Corrective feedback is according to the authors associated with the constructivist perspective of learning, whereas the reinforcing feedback more often is given by those who think that learning is the result of associations between stimuli and responses. Both types of feedback can be helpful, and different kinds of feedback may be more effective for different kinds of learning.

2.4 Automated assessment

Pears et al. (2007) say the following about automated assessment: automated assessment tools can benefit both teachers and students and are something that almost is mandatory if one has many students. If one has many students, the workload of correcting assignments can become unmanageable and by solving this by limiting the number of characters will not benefit the students learning processes. Therefore, it is good to use automated assessment made by a computer.
A survey of automated assessment approaches for programming assignments has been done by Ala-Mutka (2005). Because of the nature of programs, she states that they are a good subject for automated assessment and help the teachers concentrate their work on issues that need interaction with students more. The previous research shows for instance that many teachers mainly see the possibilities of computer-aided assessment as multiple choice questions. There are several factors one can assess including functionality, efficiency, programming errors and design. One problem with automatic assessment is that it might be difficult to get a holistic perspective of the program; even if it is not possible to execute the program itself, a program could get a reasonable mark. The author states that the “most common form of assessment for programming assignments is to check that the program functions according to the given requirements” (Ala-Mutka, 2005, p. 87), and “that the functionality is usually tested by running the program against several test data sets” (Ala-Mutka, 2005, p. 87).

There are two major methods for analysing code. Dynamic analysis means that the program is executed, as the previous example where the program is run against several test data sets. Static analysis means that evaluation can be done by collecting information from the program without executing it. According to Ala-Mutka (2005), this has historically been popular and also gives opportunities to detect issues that would not have been covered by the test cases. Also, the static analysis does not need the program to be executable, even if there are compiling errors the static analysis can be performed.

The author also discusses semi-automatic versus automatic assessment and states that “teachers often agree that it is not possible to assess automatically all the issues relating to good programming” (Ala-Mutka, 2005, p. 94) and that automatic feedback may not be of a quality as high as if given by an instructor. As Ala-Mutka (2005) concludes, there are several advantages with automatic assessment such as immediate feedback, less workload for teachers and the possibility for absolute objective assessment. Nevertheless, she also concludes that the tools often are specialised towards a specific type of assignments and hence cannot be shared between universities.

Truong et al. (2004) have developed a framework for static analysis of students’ Java assignments by converting both the written solution and a model solution to XML (eXtensible Markup Language) using an AST (Abstract Syntax Tree) and then comparing them. XML is a
markup language where one can define their own tags to distinguish between different types of content. An Abstract Syntax Tree is a tree representation of the source code, which is explained in more detail in Chapter 5.

Phothilimthana and Sridhara (2017) have developed a hint system for an introductory course in the programming language Scheme\(^3\), where they have categorised potential errors as syntactic, semantic or missing cases. They are extending the mutation-based approach, where the student’s program is mutated until they receive the model program that the instructor has conducted.

J. English and English (2015) have investigated the automated assessment system Checkpoint and concluded that automated assessment made the students more motivated because of the direct feedback and the possibility to try until they succeeded with the task. The system uses both multiple choice questions and free text questions, where the latter are assessed through by “compiling and running submitted code fragments and checking functional correctness, using a style checker to assess stylistic aspects, or using timing tests to assess efficiency” (J. English, 2006, p.337).

2.5 Ontology and epistemology in Computer Science Education

Constructivism is a theory that claims that knowledge is constructed by the student as an active process, which is a result of adaptation to the surroundings. This construction builds recursively on the knowledge already possessed by the student, which means that this is an individual process and that each student will build her or his own perception of the current topic. A consequence of this is that the knowledge constructing process will most probably result in misconceptions.

Piaget (2008) talks about two key processes which result in intellectual growth, assimilation and accommodation. Assimilation is about using existing schemas to handle new topics, whereas accommodation is when the existing schema is not working, and thus the learned behaviour is not adequate and has to be changed in order to deal with a new situation. When these two processes are in an equilibrium the

\(^3\)https://www.scheme.com/
Figure 2.1. A simplified learning curve in terms of assimilation and accommodation. Assimilation is illustrated as an activity performed on the current mental model held by the student and accommodation is a leap that requires the model to change. The knowledge or amount of information learned is not illustrated by the figure. Figure by the author.

schemas of the student are altered. Also, the wish of reaching equilibrium can be considered as the force which drives the learning process since one does not like the frustration caused by the unbalance between assimilation and accommodation. With this view of learning, one can say that learning is not a steady process but rather consists of leaps of different magnitudes, where the students are altering their models in different ways. This is illustrated in Figure 2.1. Please note that the assimilation and accommodation processes also might lead to less usable models at first, this is only a simplified model.

Figure 2.2 tries to visualise the concepts in the constructivist perspective of learning. After assimilation the students reach equilibrium and when they encounter a new situation, either their models are usable enough to assimilate or they reach a disequilibrium. If they reach
Figure 2.2. Learning process with constructivism concepts. Figure by the author, inspired by McLeod (2015).
a disequilibrium, they have to accommodate in order to change their models into more usable ones.

A significant concept in the constructivist theory is that one cannot transfer knowledge directly from teacher to student; but rather let the student build their knowledge and then as teachers check whether the built models are viable or not. This also means that the teacher cannot ignore the students existing knowledge since this is what the new knowledge is based upon. Instead, the teacher has to ask questions to the student in order to identify and understand the current theory used by the student. This to obtain sufficient information to be able to guide the student to the “correct” theory, and therefore help the student with the accommodation process.

Consider Figure 1.1, which illustrate the intersection between the constructivist theory of knowledge and the formative assessment. The teacher is performing one of the key processes in formative assessment: establishing where the students are in their learning process. If something in the student’s models seems to be incorrect, the teacher has to conclude what is wrong to determine a new destination, which is a more usable model. This is the second process. The response given by the teacher is later one step towards the third process: establishing what needs to be done to get them there.

Misconceptions are not something bad or at least not a mistake, but rather a logical construction based on a schema held by the student. As a teacher, one must identify the misconception that the student is holding and then understand the underlying model that caused the misconception in order to be able to help the student change the model into a more viable one.

The computer science student is faced with immediate and brutal feedback on conclusions drawn from his or her mental model. More graphically, alternative frameworks cause bugs (Ben-Ari, 2001, p.57)

Comparing to physics, where one has to wait a week to get feedback on an assignment only to see that one has misunderstood something, the consequences of misconceptions are often exposed immediately when programming (Ben-Ari, 2001).

Some consequences drawn from the constructivist perspective of learning computer science are as follows. The choice of language is not arbitrary since poor language design or implementation can result
in difficulties building a viable model of the computer. Furthermore, one should wait with teaching object-oriented programming since the novice students “do not have the cognitive framework to grasp the concepts underlying object-oriented design, because they have no experience dealing with types and functions, much less classes, function members or inheritance” (Adams in Ben-Ari (2001, p. 62)). Due to the absolute ontology provided by the computer, one could claim that programming exercises should be delayed until a good model of the computer has been constructed by the student. Another view is that one should use some kind of trial-and-error methodology to learn programming, something that can delay the process of constructing a viable model as well as being unprofessional later. In relation to the trial-and-error methodology, Wiliam and Leahy (2015) say that the feedback given should focus more on the longer term. Using only trial-and-error may help the student pass the test but have negative consequences for the long term. Another consequence of the constructivist perspective of learning is that closed labs should be preferable from a constructivist viewpoint, which is supported by empirical evidence discovered by Thweatt (1994).

2.6 Compiler Error Messages and Learning

One interesting question is how the compiler error messages can help the novices in their learning process. The compiler error messages might be a good source of feedback and also have an impact on altering the student’s mental models. Nienaltowski, Pedroni, and Meyer (2008) investigate what can help novices by considering a number of different compilers for five different languages. The messages produced by the compilers are put in three different categories and explore those in relation to error type, performance and response time. Students from introductory programming courses answered multiple choice questions and the results are that more detailed error messages do not necessarily increase the understanding of errors. Instead, more important factors are how the error messages are structured and where the information is placed. Of the different hypotheses tested in the study, they conclude that higher experience leads to faster answers, but it is difficult to say whether more information leads to more correct answers or shorter response times.
A study by Prather et al. (2017), using mixed-methods experiments, shows that students do read the Compiler Error Messages and that Enhanced Compiler Error Messages are found more helpful in general by students. By the word *helpful*, the study says nothing about helping the student towards a more usable model. Instead, the helpfulness is measured on whether the student posts a successful code as a result of an error message. The study also concludes that the Enhanced Compiler Error Messages also lead to fewer misunderstandings of the messages.
Chapter 3

Methodology

This chapter contains a description of the methods used in this thesis. The chapter discusses:

- The method used to obtain information about common misconceptions among novice programmers.
- The development of the Artificial Teaching Assistant.
- The evaluation of the Artificial Teaching Assistant.
- Ethics, sustainability, and societal relevance related to this thesis.

3.1 Development of a theory about common misconceptions among novice programmers

To be able to conduct the software that should help students with their coding by providing formative assessment, the knowledge of the common misconceptions is required as this is what should be identified by the software. To obtain information about common misconceptions among students, a phenomenographic study is done to get an understanding of active teachers’ perceptions of the problems by interviewing five active teachers at KTH. Phenomenography is a theory of inquiry where direct experience is the key factor (Cohen, Manion, & Morrison, 2007). It is worth noticing that one might say that phenomenography is associated with description rather than analysis.
Table 3.1

<table>
<thead>
<tr>
<th>Teacher</th>
<th>Experience</th>
<th>Programming languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10 years</td>
<td>Python, Java, C++, Go</td>
</tr>
<tr>
<td>B</td>
<td>30 years</td>
<td>Python, Java</td>
</tr>
<tr>
<td>C</td>
<td>15 years</td>
<td>Java, Python</td>
</tr>
<tr>
<td>D</td>
<td>20 years</td>
<td>Java, JavaScript</td>
</tr>
<tr>
<td>E</td>
<td>2 years</td>
<td>Python</td>
</tr>
</tbody>
</table>

(Denscombe, 2016), but because of the scope and size of this study, the belief is that this is the right way to go to identify some problems that students might have.

One alternative would be to analyse old exams in order to identify common misconceptions. However, the area of interest here is not the misconceptions after the course but rather misconceptions during the course. Instead of visiting lab rooms and asking the students about their misconceptions, the aim is to ask teachers with years of experience from different student groups, to achieve a more heterogeneous user group.

The main part of the informants was chosen by first emailing one of the authors own contacts at the School of Electrical Engineering and Computer Science at KTH, who provided a list of possible candidates with several years of experience. In total, nine individuals were contacted, and interviews were scheduled with four of them. The others did not respond. One more teacher was added via other contacts. The interviewed teachers all were employees at KTH, many of them with decades of experience in teaching introductory courses in programming. The study itself is independent of which language the courses used, but many of the teachers had experience in teaching Python during the latest years and Java before that and some used primarily Java and JavaScript as illustrated in Table 3.1. The interviews were held in alphabetical order.

The interviews were semi-structured, where the result of each interview could act as input to the next. Each informant was informed of the purpose of the study. After approval of the informant, the interview was audio recorded and later analysed. This was done through a content analysis (Cohen et al., 2007) by extracting the main topics that were discussed by searching for common themes in the material. The
content analysis was a qualitative conventional content analysis, since this is generally used when the purpose is to explore or investigate phenomenons (Hsieh & Shannon, 2005). Here, predefined categories are avoided. Instead, they are created as the material is analysed. The data analysis was conducted as described by Hsieh and Shannon (2005). First, the material was listened to as a whole. Then, the main points from each interview were noted. After that, the material was analysed through highlighting similar words or themes. When something was unclear, the recording was listened to again to clarify the notes. It is worth noticing that the themes come from the author’s interpretation of the interviews and are not based upon any specific theory.

The following questions were originally considered, and later on, examples from earlier interviews were added to see whether more of the teachers experience the same problems even if they did not come up with the example themselves. Some problems were recognised when mentioning them, whereas some stated that they have not experienced that kind of problem. This probably means that those questions did not affect or bias the interview.

- Which student groups do you teach?
- Which problems have you noticed among the students?
- Which are the most common misconceptions among the students?
- Do you have any examples of incorrect code or other things that the students do incorrectly?

3.2 Development of the Artificial Teaching Assistant

According to Comer et al. (1989), the discipline of computing can be divided into three different paradigms: theory, abstraction and design. The theory paradigm shares a lot with mathematics and logic and works according to definition, theorem, proof and interpretation. The abstraction paradigm, commonly known as the scientific method, gains knowledge through induction by forming a hypothesis and constructing a model which leads to an experiment design, data collection and
finalises with result analysis. Finally, the design paradigm is the engineering way of working where the requirements are stated, followed by designing and implementing the system and finalised with testing. In this thesis, the following approach is used (based on Comer et al. (1989)):

1. Narrow down the identified misconceptions to a reasonable amount for this scope.

2. Construct code examples that correspond to these misconceptions.

3. Design and implement a software that detects these problems in code.

4. Evaluate software towards a larger set of test data.

5. Discuss the results.

This thesis has its foundation in the design paradigm, where the initial investigation of common misconception states some of the requirements. The abstraction paradigm is also used in solving the research problem since we have an inductive approach. Under the premise that the code examples correctly reflect the identified misconceptions, and shown that these are detected by the software, we can draw the conclusion that the software correctly detects the identified misconceptions.

### 3.3 Evaluation of the Artificial Teaching Assistant

The evaluation is done by creating 5 programs, which are sent to 5 persons. Those persons were collected by asking both students at the master of science in engineering and in education programme, as well as asking people working with programming. By choosing students as error inserters, the aim is to have participants that more recently learned programming and therefore might have a greater understanding of difficulties they have had. Though asking participants working with programming, the aim is to have people from other disciplines and with other views of coding, to prevent a too homogeneous group.
Each participant is given the same information, which asks them to insert one or more problems into the provided program code examples. Either the participants can choose to insert errors based on their own perception of common misconceptions or open another text file with the results of the investigation of common misconceptions attached or choose to do both tasks consecutively. By doing this way, the errors inserted will most probably consist of the errors that should be detected by the ATA. Furthermore, by first inserting errors based on their own perception, errors not detected by the ATA hopefully will be inserted as well. This will lay a foundation for later discussion.

Since the participants sometimes inserted more than one error, and some errors were parse errors (see further Chapter 6), the programs holding more than one error were split into several files. Then, the evaluation is done by running the ATA towards each program. The characteristics of the error is noted in a spreadsheet with the note about whether the positioning is correct or not.

The participants could have been asked to create the programs themselves as well. This would have required them to dedicate more time for the task, something that is difficult to ask for. Furthermore, by having the same program examples as a starting point, the manual identification of the errors are easier to do.

### 3.4 Ethical aspects, societal relevance and sustainability

The first part of the thesis, the investigation of common misconceptions among novice programmers, and the inserting of errors, are the only parts that contain interactions with humans. During the work with this thesis, the four main requirements provided by the Swedish Science Council regarding research ethical principles have been accounted for (Vetenskapsrådet, 2002). These are the information requirement, the requirement of consent, confidentiality obligations and the utilization requirement. Cohen et al. (2007) list questions that should be taken into consideration when interviewing. The topics of the interviews are not of the kind that they could hurt the interviewed persons nor their supervisors. All the interviews are recorded after orally given consent from the interviewee and the results are shared with the interviewees to prevent different interpretations.
It could be worth mentioning that the Artificial Teaching Assistant will be used by people, and that those interactions have to be supervised by someone or at least have someone responsible and that the responses are formulated in such a way that the message is clear, and the possibilities of misunderstandings are reduced as much as possible. Also, another topic of consideration is replacing humans with computers in one way or another. This is not a concern at the moment when we are talking about the ATA since it grows from a need where we do not have enough teachers, but as with all automation, there is a risk that we replace job opportunities in the future.

Since the outcome of this thesis is a technical artefact that helps students in their learning process when learning programming, one could claim that it has high societal relevance. Furthermore, programming is one skill that is argued to be important, both for the society as well as for the individual. One example of this is the addition of programming to the Swedish curriculum (Regeringskansliet, 2017). One sustainability aspect of this is that the need for transportation is decreased since the student does not have to be at the university in order to participate.
Chapter 4

Results from the Investigation of Common Misconceptions

This chapter contains a description and analysis of the results of the interviews to identify common misconceptions and problem areas in introductory programming. Specifically, the chapter covers:

- Interview A
- Interview B
- Interview C
- Interview D
- Interview E
- A summary and short analysis of the interviews

4.1 Interview A

The informant is an experienced teacher in languages like Python, Java, C++ and Go. The identified misconceptions and problems are:

- Pointers.
- Object orientation.
- Recursion.
- Scope: local vs. global variables.
• Assignment operator.

• Variable types.

According to the informant, what they have in common is that the students do not have a correct mental model of the computer memory. The students may be very skilled in looking at code and identifying the order of which a set of instructions will be executed or finding out what an expression will be evaluated as, but they do not have a mental model of the memory.

Although pointers is not a covered topic in the introductory courses, references have a similar behaviour when working with objects and arrays for instance.

Other misconceptions include thinking of variables as the same thing as a variable on a calculator; the latest written value is what is contained in the variable. This is true in many cases, but not when the scope or references with different names that point to the same object are considered. Another misconception is how floating point numbers are handled, and the fact that $0.1 + 0.1 + 0.1 \neq 0.3$. The informant also added that the assignment operator is a problem for novice programmers with regards to the similarities with algebra; for instance, that $x$ is an unknown value and as soon as one gets to know what the value is, it is what it is, or that one could write $3 = x$ as well as $x = 3$. The informant states that mathematics can be helpful when learning to program, but it is not the complete truth. Apart from syntax errors in form of missing parenthesis, one common coding error is type errors.

### 4.2 Interview B

The informant has worked with introductory courses in programming for many years, both in Python and Java. Some of the misconceptions and problems identified are:

• The variable concept.

• Variable types.

• The difference between the if-statement and the while-loop.

• Functions: calls, definitions, parameters and return values.
• Sequences.
• Local and global variables.
• Indexes and values in arrays.
• References.

The informant believes that some students have difficulties with distinguishing between the if-statement and the while-loop. Not because of what they do but rather their similarities in appearance since both have a condition but one of them is executed one time and the other multiple times.

There are many concepts regarding functions that students find problematic. These include concepts as parameters, return values, local and global variables, definitions and function calls. Students have troubles with the placeholder function of parameters and want them to have a value as well as letting the variables they pass into a function have the same name as the parameters. Another problem is that the students forget to return a value from the function or does not handle the returning value; they just call nonVoidFunction(). Furthermore, some students have troubles with sequential thinking and that some statements have an irreversible effect on the state of the program.

In addition, students tend to forget to call their previously declared function. When it comes to arrays, some students struggle with the difference between an index and the value, which can be even more problematic talking about maps that can have strings as indexes for instance. At last, students are struggling with the differences in assignment between for instance integers and arrays. If we have integers $x$ and $y = 5$, we can write $x = y$ and now both $x$ and $y$ contain the value 5, but if we have an array we will not have two different arrays with the same content.

4.3 Interview C

The informant has taught programming in both Java and Python for at least 15 years, mostly focusing on Engineering Physics students. Variables are a misconception from earlier interviews that were forwarded into this interview since the informant did not bring that up. The informant did not recognise that as a problem. Identified problems are:
• Parameters and arguments.

• Sequence in loops.

• Extensive use of if-statements.

• Recursion.

Some students are writing statements in a loop that does not belong there or are using if-statements when it is not necessary. The informant claims that in some languages one can change the controlling variable in a loop, and in some languages, one cannot, which can be a problem. The informant concluded that students have problems with parameters when it comes to functions, we have a name for something, but it does not have a value, with the common misconception that the variable used when calling a function must be named the same thing as the parameter. This is not as problematic when you have taken courses in mathematics and know about mathematical functions, but for students that do not have the mathematical background, or when teaching young children, this can be more problematic. The interviewee states that it can be even more problematic when the students are asked to alter the arguments with some value, which leads to a call like `someFunction(value + 1, value - 1);`.

4.4 Interview D

The informant is currently working with the second and third programming courses at a programme at KTH, hence not the first introductory programming course. Because of this, many of the common problems occurring for novice students are unknown for the informant since the students either have managed to learn the common misconceptions or have not passed the course and therefore not started these courses yet. The courses where this informant teaches are in Java and JavaScript respectively. Identified problems are:

• Scope.

• Function call trees.

• References to functions and arrays.

• Extensive use of if-statements.
• Break out of loops.

Some students have difficulties with the scope and where they can access certain variables. In addition to this, by omitting the `var` keyword in JavaScript, the variable becomes global instead of local independently of its position. In later courses when the complexity or the size of the written programs increases the difficulty level also increases. Also, when having nested function calls, i.e. calling functions that call other functions, some students have difficulties with keeping track of values and where they are in the chain. When it comes to JavaScript, the informant mentions that anonymous functions and callbacks can be potential areas of problems as well as pointers and references to functions and objects. Furthermore, the informant mentions that some students tend to nest if-statements or have an extensive use that is not necessary. Finally, many students do not use concepts like `continue` and `break` in loops or forget that they can return inside a loop. Instead, they use flags and if-statements. The informant recognises problem with references when it comes to function, arrays and pointers when this is brought up as a result of the previous interviews.

4.5 Interview E

The informant is a PhD student who teaches introductory courses both on campus and online, where they use the language Python. Identified problems are:

• Objects and classes.

• Return values.

• Assignment operator.

• Loops.

• Scope and variable names.

Many students have problems with starting to program, which includes everything from opening the editor to figure out where to begin coding. Furthermore, objects and classes are the first really abstract thing that the students encounter and with this the problem of distinguishing between functions and methods come about. Return values
are another problem area, where students tend to miss to catch the value. This is illustrated by the students writing a function and then drawing the conclusion that it is not working because of the lack of results when in fact they have forgotten to do something with the returned value from the function. Also, many students have problems with the asymmetry of the assignment operator and this in connection with functions since they want the function first to be executed and then catch the value into a variable, we are writing the code from left to right, but the execution is rather right to left.

When it comes to loops some students have difficulties with identifying which code that should be inside a loop and what should be outside, for instance, print statements. Furthermore, students have problems with breaking out of the loop by using `break` or `return`. It is more common to use flag variables inside the loop and if statements after the loop. The informant also mentions that base cases can be difficult to identify and implement.

Many examples are using the same variable names in many places in different scopes and it is common that students choose to call their parameters to a constructor the same things as the attributes later, which means that the difference between the parameter and attribute (name) is not clear.

Because of the dynamic typing in Python, students have problems with the comparison of values. One common example is to have an input `age`, which is a string, and then compare it to an integer only to find out that the comparison always is false. Another consequence of this is problems with the concatenation of strings and integers or addition between the same.

### 4.6 Analysis

As illustrated in Table 4.1, interviews A, B, D and E talk about the difficulties with local and global variables. The problems with the assignment operator are mentioned by informant A, B and E, such as its asymmetry as well as the difference from algebra where the objective is to know a specific value but here the objective is to assign it. This means that the students’ models from mathematics are not directly applicable to programming, and hence have to be altered in order to work with this new context. Such an alteration process is an example
where the student has to accommodate.

Functions is another subject that concerns a lot of students, both handling function parameters and arguments but also calling them and catching their return values. All informants express that functions can be a problem in the introductory courses. Many of the problems could be considered as trivial, such as forgetting to call a declared function or using the returned value of a function. However, by noticing those gaps in the students’ models many problems and irritation moments can hopefully be reduced. This is an example of assimilation, where new knowledge is added to an existing model.

Next topic of concern is control-sequences and loops. Students tend to use a lot of if statements as seen in interviews C and D, or use if-statements and flag variables instead of break statements as seen in interview E. Also, interview D claims that students have problems with getting out from loops. In addition, a few of the informants say that recursion is a problem. The reason that everyone has not expressed this concern is probably due to that recursion is not a topic for all introductory programming courses.

Finally, some of the informants say that variable types and the variable concept are a problem, whereas some did not say that it is a problem. This could be dependent on which student group you teach since the more mathematical-oriented programmes may have easier to grasp the concept since it is more familiar, in comparison to other programmes. Hence, some students may have to assimilate whereas others may have to accommodate in order to learn these concepts.

Consider Table 4.1. Teacher B is the one with the highest amount of identified difficulties. This could be due to several things, either the informant is very aware of their teaching or has very low thoughts about their students. One other possible reason could be that the person is teaching at programmes where the frequency of programming experience is lower and hence more novice programmers, along with a good communication with the teaching assistants and a high presence in the computer labs.

One surprise is the low frequency of mentions about variable types and the variable concept. As shown in the study by Du Boulay (1986), it is common with misconceptions regarding these topics. However, it not a big issue for the informants. Furthermore, it was unexpected to have that many misconceptions related to functions, but as explained in the interviews it is a way of altering the sequential timeline of the
Table 4.1

Identified difficulties

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Function parameters</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Function calls</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointers and references</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function return values</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Extensive if usage</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Break out of loops</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Recursion</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence in loops</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Objects and classes</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable types</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The variable concept</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

program execution and therefore something that is not as intuitive as one might think at first.
Chapter 5
Design and Implementation

This chapter describes the design and implementation of the Artificial Teaching Assistant. The chapter covers:

- The design of the ATA.
- The implementation of the ATA and the conclusion to create the language Javascriptish.
- Different characteristics of the identified misconceptions.
- The implementation of the misconception detection.
- The implementation of the misconception reporting.

5.1 Design of the ATA

Since the students want to learn programming languages they can use after graduation (Pears et al., 2007), one should not use a specific language only for education. This makes Python and JavaScript more suitable alternatives for a course in introductory programming and Java less suitable because of its complexity and object-oriented manner, which also is supported by the constructivist theory of learning. Since the ATA is supposed to be used in MOOCs online, the decision was taken that JavaScript is the better language of the two because of the fact that it can be interpreted in the browser.

The vision of the MOOC in fundamental programming is to include some constructs that are not implemented in JavaScript, for instance, tuples and algebraic data types as in the programming lan-
guages OCaml and Haskell. Therefore, the used programming language is Javascriptish, a language developed by the main supervisor. Javascriptish is at the moment a pure subset of JavaScript (with the exception of printing), but with the aim to extend the language with constructs that could be suitable for a course in fundamental programming.

To be able to run code written in Javascriptish, it is translated into JavaScript. This process is illustrated in Figure 5.1. This is done by first doing a lexical analysis, where the code is translated into tokens which are the smallest meaningful units of a program. Afterwards, these tokens are parsed to a parse tree (Scott, 2006). This parse tree is an abstract syntax tree (AST) that contains the structure of the program, which can be printed as JavaScript for later execution. When implemented in an online course, the AST will first be analysed by the ATA and if successfully evaluated it will be sent to the pretty printer (PPrint), which prints the syntax tree as JavaScript that is saved in a temporary file and then executed by Node.js. As the implementation is built now, the user can choose to either run the program or analyse it via the ATA.

There are both concrete and abstract syntax trees. A concrete syntax tree is an exact representation of the source code, which means that there exists a mapping from the context-free grammar that defines the language into the tree. This means that the keywords of the language are preserved, for instance else if in Java or elif in Python. Also, to prevent ambiguity, the concrete syntax tree contains information about operator precedence by nodes at multiple levels. The abstract syntax tree could be seen as a simplification of the concrete syntax tree and contains the information that is needed to represent the meaning of the program.

The objective of this thesis is the implementation of the ATA. The other components in the chain, as shown in Figure 5.1, are implemented by the main supervisor.

The lexer and the parser are built on OCaml’s version of Lex and Yacc (Levine, Brown, & Mason, 1992), which are two standard Unix tools for building compilers and interpreters. Yacc is an LALR(1)-parser, which stands for Look-Ahead left-to-right, rightmost derivation. The number one means that the parser is looking one token ahead, which is used to resolve differences between different rule patterns during the parsing process.

As mentioned in the related work chapter, one can do both static
The design of programming languages includes two main concepts: syntax and semantics. 

The syntax of a programming language is the part of the language definition that says how programs look: their form and structure. [...] The semantics of a programming language is the part of the language definition that says what programs do: their behavior and meaning. (Webber, 2003, p. 10)

The semantics can be divided into static semantics or dynamic semantics. Dynamic semantics is about what actually happens when the program is executed. Static semantics is the part that can be considered at compile time and includes topics as whether all variables are declared, typing and if functions are called correctly.

The majority of the identified problem areas among students have to do with the structure of the code and the static semantics, which makes static analysis suitable. With this said, all problems might not
be able to be detected using this method, and other program features as efficiency and correctness might not be measurable. Therefore, it could be appropriate to add these analysis methods further on.

### 5.1.1 Requirements

Among the common misconceptions that are identified in the previous chapter, the following are the ones that will be in focus. This since they can be identified with static analysis and belong to the contents of a course in fundamental programming.

The Artificial Teaching Assistant should detect problems related to:

- Variables in different scopes.
- Usage of function parameters.
- Calling functions.
- Catching return values.
- If usage inside loops and breaking out of loops.

### 5.2 Implementation of the ATA

Since one topic of the investigation in this thesis is the grammar of programming languages, a language where this can be expressed in an efficient way is needed. Therefore, the language OCaml is used. OCaml is a programming language with both imperative and functional features. Furthermore, the expressiveness of the language along with its algebraic data types makes it very straightforward to declare the grammar of the language and traverse through the generated syntax tree. This means that it is possible to declare all the variants included in a programming language in a straightforward way, and then use those variants to create a tree structure that corresponds to the structure of the code.

Javascriptish is designed by the main supervisor David Broman, who has written the lexer, parser, AST definition and code generator. The author focuses on the function that analyses the code written by the student and detects possible misconceptions and contributes to the implementation of the language by adding support for `break` and parts of the modulo operation.
The language Javascriptish itself consists of different terms, for instance, an if, while or function call. The mission is to write functions that traverse through the tree of all these terms that build a program and detect patterns that indicate the defined problems.

### 5.2.1 Defining Javascriptish

The language is divided into different types of terms. As seen in Listing 5.1, the language is divided into 13 different terms, or tm for short. The listing is an excerpt from the definition of the Abstract Syntax Tree, written in OCaml.

```ocaml
type tm =
  | TmDef of info * isconst * ustring * tm
  | TmWhile of info * tm * tm
  | TmIf of info * tm * tm * tm option
  | TmAssign of info * tm * tm
  | TmRet of info * tm
  (* Expressions *)
  | TmVar of info * isconst * ustring
  | TmConst of info * const
  | TmFunc of info * ustring list * tm
  | TmCall of info * tm * tm list
  | TmBreak of info
  | TmProj of info * tm * ustring
  | TmArrIndex of info * tm * tm
  | TmArray of info * tm list
  (* Other *)
  | TmScope of info * tm list
```

*Listing 5.1.* The different terms in Javascriptish

In each term, there is an info item that consists of the file name and row and column for the starting point and ending point of the term that is used to provide information about the location of the errors. Furthermore, the terms usually contain a pointer to the next term in the tree, or a list of terms in the scope for instance. The different constructors that together build up a type, such as tm, are called variants.

### 5.2.2 Characteristics of the misconceptions

One of the most common problems identified by the interviews were about the handling of variables over different scopes. This is seen in
for instance, Listing 5.2, where the variable \( y \) is declared inside the body of the if-statement and hence is not accessible when the variable is printed outside the scope. This can be analysed by checking for all variable usages that a variable definition exists in the current scope or a scope higher up in the tree structure.

```
1 var x = 42
2 if ( x > 7 ) {
3   var y = 35
4 }
5 print(y) /* y is not defined here */
```

Listing 5.2. Variables over different scopes

Another common problem is regarding function parameters. As illustrated in Listing 5.3, the function is called with the wrong number of parameters. In addition, the interviews showed that many students think that they have to name the variables passed as arguments in the same way as the parameters, which could be added as a note to the user in order to tell them that it is not mandatory. This is illustrated in Listing 5.4.

```
1 function three_sum(x,y,z) {
2   return x + y + z
3 }
4 print(three_sum(1,2)) /* three_sum is called with two instead of three parameters */
```

Listing 5.3. Wrong number of function parameters

```
1 function add(val1,val2) {
2   return val1 + val2
3 }
4
5 var val1 = 1
6 var val2 = 2
7 var added = add(val1, val2)
```

Listing 5.4. Naming the arguments in the same way as the parameters

Sometimes, when students declare their own functions, they forget to call them. This leads to unnecessary trouble for the students when they try to debug their code. Listing 5.5 illustrates this problem.

Just as forgetting to call their functions, return values sometimes are forgotten too. This is illustrated in Listing 5.6.
1  function add(a, b) {
2      var sum = a + b
3      print(sum)
4      return sum
5  }
6
7 /* We have declared our function add, but it is not called */

Listing 5.5. The student has forgotten to call their declared function

1  function add(a, b) {
2      return a + b
3  }
4
5  add(123, 64) /* the function is called, but the returned value is not caught */

Listing 5.6. The return value of the function is not used

To terminate a loop there exists a keyword called break. According to the interviews, students tend to use a flag variable, that is a boolean variable, which is set to true if a certain condition is met during the iteration of the loop. The flag is later checked in another if statement after the loop. Instead, they could use break or return to prevent unnecessary runs of the loop. This problem is illustrated by Listing 5.7 and is characterised by a boolean variable inside the body of the if-statement, and then the same boolean variable outside the while loop.

1  var i = 10
2  var done = false
3  while (i > 0) {
4      if (i == 5) {
5          done = true
6      }
7      i -= 1
8  }
9  if (done) {
10     print(i)
11  }

Listing 5.7. Not using the keyword break

To not use the break keyword is not an error, this is about better code design. Using the keyword both makes the code more readable
as it is easier to see what the code does without having to execute it manually reading it. Furthermore, preventing unnecessary runs of a loop gives better performance.

5.2.3 Detecting misconceptions

The input to the analysing function is the term that is the root of the AST, and the output is a list of environment_info, of the variant ErrorMsg. It is a recursive function, which matches the provided term with the different variants and does different things depending on the type. If the current term has a child term, the function is called once again with that term as the root of the tree. This leads to a depth-first search of the abstract syntax tree.

```ocaml
type environment_info =
  | ErrorMsg of message
  | VariableInfo of info * ustring
  | FunctionInfo of info * ustring * int * bool * bool * ustring list
```

Listing 5.8. The type environment_info

The analysing function also takes one additional argument, which is an accumulator that holds information about the environment; error messages, defined variables and defined functions. The accumulator is implemented as a StringMap, where each key is a string and the values are lists of environment_info. The list with VariableInfos is cleared visiting a parent scope, whereas error messages are preserved in the accumulator in order to be listed after the analysis has terminated. Thus, three different variants of environment_info are constructed as shown in Listing 5.8. The message type is defined as shown in Listing 5.9, and consist of an id, the severity of the problem (WARNING/ERROR), info about the position of the error, and additional arguments to the message. A VariableInfo consists of info about the position, and the name of the variable. The FunctionInfo also takes info and name of the function, but also the number of arguments, a boolean which tells if the function is called or not, a boolean that tells us if the function returns a value or not, and the list of parameter names as well.

```ocaml
type message = id * severity * info * arguments
```

Listing 5.9. The type message
Variables over different scopes

Every time the ATA encounters a definition that is a variable, a 
VariableInfo is added to the environment. Each time the ATA en-
counters a variable term, it checks whether that name exists in the en-
v

vironment or not. If not, the ATA would like to suggest another vari-

able name for the case that the student has misspelt the variable name.

In order to suggest alternative variable names for the student, the 
Levenshtein Distance algorithm is used (Haldar & Mukhopadhyay, 
2011). The algorithm measures the similarity between two strings by 
calculating the number of insertions, deletions or substitutions that are 
required to transform one string into the other. The algorithm is im-
plemented as shown in Listing 5.10. As an example, the words sitting 
and kitten have distance 3.

```
let rec levenshtein_distance s len_s t len_t =
  if len_s = 0 then len_t
  else (
    if len_t = 0 then len_s
    else (let cost = if Ustring.equal (Ustring.sub s (len_s-1) 1) (Ustring.sub t (len_t-1) 1) then 0 else 1 in
         minimum
         ((levenshtein_distance s (len_s-1) t len_t) + 1)
         ((levenshtein_distance s len_s t (len_t-1)) + 1)
         ((levenshtein_distance s (len_s-1) t (len_t-1)) + cost)))

Listing 5.10. Levenshtein Distance algorithm
```

Thus, when the ATA encounters a variable that is not defined in
the scope, it calculates the distance between the provided name and 
all defined variables and suggests the name with the shortest distance 
to the provided name. Then an ErrorMsg is created with id 
VAR_NOT_IN_SCOPE, and with the provided name and the suggested 
name as arguments.

Function parameters

The problem with providing the wrong number of parameters is solved 
in a similar manner. Each time the ATA encounters a definition that 
contains a function, a FunctionInfo is added to the list of defined
functions in the environment. Here the number of parameters provided in the function definition is calculated. Later on, when the ATA encounters function calls, it checks if the function is among the declared functions and if it is, the number of provided arguments is compared to the supposed number of parameters. If these numbers are not the same, the ATA adds an error with id `WRONG_NUMBER_OF_PARAMS` and provide the function name, the expected number of parameters and the number of provided arguments as arguments to the message.

### Calling functions

Each time the ATA encounters a function call of a declared function, it finds the corresponding `FunctionInfo` in the environment and sets the boolean, that tracks if the function is called or not, to true. When the traversing of the abstract syntax tree is done, the ATA can ensure that all declared functions are called. If not, the ATA creates an error with id `FUNCTION_NOT_CALLED` with the name as an argument.

Just as with the variables described earlier, if a call to a function that is not defined is encountered, the ATA tries to find a suggestion. Then an error is created with id `FUNCTION_NOT_FOUND`, and with the provided name and the suggested name as arguments.

### Catching return values

When the ATA finds a function definition, it also traverses down the body of the definition to see whether the function ends with a return statement or not in order to set the boolean variable that tracks if the function returns a value or not. Later on, when a function call is encountered, if the function is supposed to return a value, the ATA has to ensure that the function call is encapsulated in an assignment or in some other place where the returned value will be used. Such places include function parameters, the head of `if` and `while` statements or in `return` statements. If the result is that the returned value is not used, an error message with id `UNCAUGHT_RETURN` is created.

### Breaking out of loops

All the other error detection mechanisms are merged into one single function, but the checking for `break` usage is put into another function. This due to the special patterns that indicate this sort of be-
CHAPTER 5. DESIGN AND IMPLEMENTATION

haviour. The ATA checks for assignments of boolean variables inside an if that itself is inside a while. It that variable later is used outside the while loop, this indicates this misconception and an error with the id BOOLEAN_INSTEAD_OF_BREAK is created.

5.2.4 Reporting misconceptions

When the analysis is done, a list of ErrorMsg is returned. As the implementation looks now, the complete list is printed out to the student. This is done by converting each id into a string, where the provided arguments also are added if applicable. Each error message has information about the probable location for the error, in terms of the file name and the row and the column numbers for the start and the end of the problematic term.

As the study by Traver (2010) argues for, the error messages are tried to be kept as more user-centred. Furthermore, the aim is to make them more informative and specific as argued for:

Another disadvantage is that, since the method is automatic, messages are all of the type ‘XXX expected’, ‘XXX expected before YYY’, or ‘XXX expected instead of YYY’, which are not as specific and informative as hand-crafted ones can be (Traver, 2010, p.5).

The error messages are also made in such a way that they should be formative. The feedback should move the students forward by giving them comment-only marking (Black & Wiliam, 2009), thus no grades or similar. Furthermore, one of the key processes identified in formative assessment is establishing where the students are in their learning process. This is done by the ATA, by comparing the results to some of the skills that the students should have when finishing a course in fundamental programming. The comments are then written in such a way to try to fulfil the third process: establishing what needs to be done so that the students manage those skills.

The resulting error messages can be seen in Figures 5.2 to 5.7. For a more personalised feeling, the error messages refer to the ATA as I and the student as you.
I could not find the variable myage in the current scope, did you mean my_age?

*Figure 5.2. The error message for scope errors, with a suggestion*

I could not find the variable myage in the current scope. Remember that a variable declared inside a function is not available outside that function.

*Figure 5.3. The error message for scope errors, without suggestion*

Consider your declaration of the function NAME and your call to the same function. Something is wrong with the number of arguments provided to the function.

*Figure 5.4. The error message for errors with the wrong number of arguments*

The function NAME returns a value, but that value is never used. Perhaps you want to assign that value to a variable?

*Figure 5.5. The error message for not using return values*
You have declared a function called NAME, but you never call it. Remember to call your functions. If you do not use your function, consider removing it from your code to make it more readable.

\textit{Figure 5.6.} The warning message for not calling a declared function

You are using a boolean variable instead of a break statement. Consider changing this in order to make your code more readable and efficient.

\textit{Figure 5.7.} The warning message for using booleans instead of break
Chapter 6

Evaluation, Results and Discussion

This chapter presents the results of the evaluation and discusses them in relation to the related work. The chapter covers:

- A description of the evaluation of the ATA.
- The results from the evaluation of the ATA.
- A discussion of the results, with respect to precision and cover, the artefact’s impact on the learning process, the ATA in the MOOC context, the problem with problems, error messages and types and parsability.
- A methodology discussion.

6.1 Evaluation

The research problem is to design a static analyser that detects common misconceptions about programming and provides feedback about what, why and where. During evaluation of the ATA, what we have to do is to ensure that the detected errors are classified correctly, and that the ATA gives a correct position of the error.

To achieve a higher validity of the evaluation, independent participants inserted errors into program examples. By doing this, the results are a lot less biased in comparison to if the author had inserted the errors himself. The participants were able to choose whether they
wanted to insert errors based on their own perception about misconceptions for novice programmers, by receiving a list with the results of the study conducted in this thesis or do them both consecutively.

In total, five different programs were evaluated. These are:

- Binary search.
- Bubble sort.
- Find the smallest number divisible by range.
- List all primes up to number $n$.
- Polynomial division.

The source code for these programs can be seen in Appendix A. Each program is quite common for computer science courses and uses functions, loops, if-statements and arrays.

As discussed in section 5.1 and shown in Figure 5.1, the written code has to pass the parser in order to proceed to the ATA. Since many program examples are written without knowledge of the implementation and the identified misconceptions, a natural implication is that parse errors occur.

In total, 77 errors are analysed by the ATA and they are categorised as follows:

- Scope error.
- Parse error.
- Function not called.
- Misspelt function.
- Parameter error.
- Nothing found.
- Unknown character.
- Returned value.
- String unterminated.

Below are some examples of the characteristics of these errors
Scope errors

Listing 6.1 and Listing 6.2 show two examples of scope errors. In the first example, the `var` keyword is omitted and hence we are trying to assign the value of the function call to a previously declared variable that does not exist in the scope. In the second example, the variable `coeff` is misspelt and therefore it is not possible to find the variable in the current scope.

```javascript
function polyShiftRight(p, places) {
    if ( places <= 0 ) {
        return p
    }
    pd = degree(p) // No var keyword here
    var d = copy(p)
    var i = pd
    while ( i >= 0 ) {
        d[i+places] = d[i]
        d[i] = 0
        i = i - 1
    }
    return d
}
```

*Listing 6.1. Scope error due to no var keyword*

```javascript
if ( coeff < 0 ) {
    if ( i < pd ) {
        result = result + " - " + covfefe
    } else {
        result = result + coeff
    }
} else {
}
```

*Listing 6.2. Scope error due to a misspelt variable*
Function not called

In Listing 6.3, the call to the main function is removed from line 26:

```javascript
function find_primes(limit) {
    var i = 2
    while ( i < limit ) {
        if ( is_prime(i) ) {
            print(i)
        }
        i = i + 1
    }
}
```

Listing 6.3. Removed function call

Misspelt function

In Listing 6.4, the function name `copy` is misspelt in its definition, which makes the calls to it later throw an error:

```javascript
function cop(arr) {
    var n = arr.length
    var i = 0
    var newarr = []
    while ( i < n ) {
        newarr[i] = arr[i]
        i = i + 1
    }
    return newarr
}
```

Listing 6.4. Misspelt function definition
Parameter error

In Listing 6.5, one argument is not passed to the function call on line 132, giving only one argument.

```
130  q[nd - dd] = n2[nd] / d2[nd]
131  polyMultiply(d2, q[nd-dd])
132  polySubtract(n2)
133  nd = degree(n2)
134  }
```

Listing 6.5. One argument removed from a function call

Unknown character

In Listing 6.6 there is written only one “&” instead of two, which leads to an unknown character. This is a lexing errors because the language Javascriptish does not support the bit-wise and operation right now.

```
13  14  while (items[middle] != value & startIndex < stopIndex)
```

Listing 6.6. Unknown character due to only one “&” instead of two

Other error types

Not found are those errors where there is an error in the code, but the ATA ignores it. Returned value is when we do not use the value returned from a function. String unterminated is when a string has its initial quote mark, but no quote mark to terminate the string.

6.1.1 Results

The frequency of every error type is illustrated in Figure 6.1. About 27% are parse errors.

Information about the position of the error is provided in each error message. For most of the messages the position is completely correct, and for some, it is not and for some, it is almost correct. An almost correct position means that the distance between the given position and the correct position is less than one line. The distribution is illustrated in Figure 6.2.
Figure 6.1. Distribution of error types.

Four of the five errors that are incorrectly positioned are the same ones that are placed in the category *Nothing found*, where no message is given by the ATA and hence no position. The fifth one comes from a parse error caused by a missing “}” after a *while*-loop, with the parser pointing the error to the end of the file. This is a natural behaviour since the parser does not know where the body of the *while* should end, but a reasonable improvement could be to state that the ending “}” should occur before the next function declaration for instance. For the six errors that get an almost correct problem position, the parser points to the line after.

In Table 6.1, the precision of the positioning of the problems is listed for each error type.

**Nothing found**

In total, four errors are not detected by the ATA. These are:

- The return value of a void function is assigned to a variable (the opposite of what is being detected now).
- Two cases of wrong conditions to a *while*-loop.
- An incorrect program: too early termination of a *while*-loop.
Table 6.1

Precision of positioning. This is the data that Figures 6.1 and 6.2 are based upon.

<table>
<thead>
<tr>
<th>Error type</th>
<th>Frequency</th>
<th>Correct</th>
<th>Almost</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope error</td>
<td>33</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Parse error</td>
<td>21</td>
<td>74%</td>
<td>29%</td>
<td>5%</td>
</tr>
<tr>
<td>Function not called</td>
<td>5</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Misspelt function</td>
<td>4</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Parameter error</td>
<td>3</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Nothing found</td>
<td>4</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>Unknown character</td>
<td>1</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Returned value</td>
<td>5</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>String unterminated</td>
<td>1</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>77</strong></td>
<td><strong>86%</strong></td>
<td><strong>8%</strong></td>
<td><strong>6%</strong></td>
</tr>
</tbody>
</table>
The characteristics of these Nothing found-problems, excluding the first in the list above, cannot be detected by this type of analysis that is done by the ATA. The ATA does not know anything about the problem that the user aims to solve, which makes it difficult to check the conditions to a loop for instance. The solution to these problems would be to add dynamical analysis as well, which is aware of the expected output for certain inputs of the program and test it towards those data. In this way, the ATA would also know more about the expected behaviour of the program. This first problem in the list could be detected by the ATA in a similar manner that it is detecting the opposite case today.

6.2 Discussion

Precision and cover

As seen in Table 6.1, 86% of the given problems are correctly positioned. In addition, the ATA succeeds in giving a correct position for 74% of the parse errors, and a good hint for 95% of them, even though the code never manages to reach the analyse phase.

One result from the interviews about common misconceptions among novice programmers is how if-statements are used within while-loops. This is not tested in the evaluation since none of the participants that inserted the errors added such an error. This includes both the group that is unaware of the identified misconceptions as well as the group that received the list. However, some of the given errors that deal with conditions might be a result of a possible interpretation of that problem description. Furthermore, the zero occurrences of these sort of problems could be explained by the specificness of the nature of the characteristics of the problem. This, since the problem, requires a very particular pattern of terms in order to be triggered. It could also be discussed whether it is an actual problem and if the ATA is the best way to handle that kind of misconception. Another way of preventing that problem from happening is to be aware of the problem and design that lecture in such a way that the problem is mentioned in one way or another.
The artefact’s impact on the learning process

Compiler messages are often difficult to interpret (Traver, 2010; Nien-altowski et al., 2008; Prather et al., 2017). By using the concepts from the constructivist perspective of learning, one could say that a huge accommodation is required to understand the compiler messages, as illustrated in Figure 6.3. This makes programming difficult and increases the frustration, which might lead to drop-outs from courses and the view that programming is boring.

![Figure 6.3. Learning programming with an ordinary compiler. The accommodation required to alter the less usable model held by the novice programmer in order to be more skilled, i.e. understand the message. Figure by the author.](image_url)

One result of the ATA is that the leap required to build a more usable model is decreased, as seen in Figure 6.4. Apart from the lesser magnitude of the leap, the ATA also allows for a longer assimilation process. This might make students feel in a broader sense of flow, because of less discomfort since they are in equilibrium and are therefore having more fun.

Basically, one can say that the ATA lowers the threshold for the student to learn programming. Is the learning faster or slower? One
could argue that the learning process is initially faster for two reasons: the ATA gives the student the ability to move forward on their own in a faster manner, and it takes less time to interpret the messages given by the ATA. On the other hand, it is hard to say what impact the ATA will have on the step towards the “real” compiler messages. Furthermore, one could argue that only encountering the ordinary compiler messages might be better, since once passing the huge threshold of mastering the messages, the learning curve gets a great start. With that said, the main target group is not those who master programming, but rather those that find programming difficult and where we would benefit from making it as easy as possible.

As discussed in Section 2.3 about assessment, formative assessment is not only about the activities, but rather the whole philosophy. The feedback given by the ATA could be argued to help the students in their own learning process, but further contributions should be added in order to shape the future of the students’ learning processes. By collecting data about the error messages and therefore the probable misconceptions, it is possible to visualise the progression for both the individual student as well as for the class as a whole. In such a case,
the data can also be used as feedback for the teacher about the progress in the course as well to improve the exercises and lectures. It would be possible to add quizzes, where the questions could be excerpts from the student’s code and the answers examples of what that could be wrong in the code.

On the topic of feedback, we can define corrective and reinforcing feedback (Wiliam & Leahy, 2015). The feedback given by the ATA is corrective since it points out the errors made by the student. However, programming could be considered as the most reinforcing feedback of them all. This since programming is very rewarding as an activity when a successful compilation of a program results in the correct output.

As mentioned in Chapter 2.4, Ala-Mutka (2005, p.94) states that “teachers often agree that it is not possible to assess automatically all the issues relating to good programming”. This thesis does not claim that the ATA should be able to automatically assess all issues relating to good programming, but rather be a step in the investigation of what is possible to automate. With the help from the ATA, the students should be able to continue to work with their programming assignments on their own, without the presence of a teacher in the room. Thus, the ATA should be able to replace some of the tasks that the teaching assistant has when working in the lab rooms at a university for instance. Ala-Mutka (2005) also criticise automated assessment as the tools seldom can be shared between universities. With the current implementation of the ATA there is nothing task-specific about what it can assess, only language dependent. Since the input to the ATA is an abstract syntax tree, it would be possible in theory to replace the initial parts of the tool chain to construct abstract syntax trees from different languages and pass them all to the ATA.

The ATA in the MOOC context

In the context of a MOOC in fundamental programming, the plan is to let the student take different routes depending on their level. This means that the students that have difficulties with some concepts can be lead to more instructional videos that explain the subject in a more detailed way, while those that grasp the concepts can move on. By doing such a thing, the current knowledge of the student is taken into consideration when designing the further learning activities, some-
thing that is a key activity of formative assessment.

One thought here is to also provide the more difficult compiler message to the student to let the student learn from them as well. In those cases where the student could benefit from knowing what an ordinary compiler would say, it would be difficult to see any problem with providing them with that error message as well. This can be something you add to the student’s journey towards mastering fundamental programming.

As concluded by Du Boulay (1986), the difficulties novice programmers encounter can be categorised into orientation, the notional machine, notation, structures and pragmatics. All of these should be emphasised in a course in fundamental programming. The notational machine and notation are the ones most central in relation to the ATA. One difference in comparison to an ordinary compiler is that the ATA provides a more straightforward interface towards the behaviour of the computer and how to communicate when the computer does not understand the given commands. Furthermore, many of the syntactic and semantic issues can be addressed using the ATA.

The problem with problems

There are two use cases for the ATA, or for the potential problems in the code. Either the problem is detected by the translator or programming language processor, with a message that is difficult to understand. Otherwise, the error is not detected by the translator and would throw a runtime error upon execution, or just not the expected behaviour. If the ATA was used, the students would instead get an error message that could help them explain why they otherwise would see an unexpected behaviour, or just know what is wrong with their code.

One of the problems detected by the ATA is detected by a type-checker in compile-time but is not detected by the JavaScript interpreter. This is when a function that does return a value is called, but that value is not assigned to a variable. This leads to a type error in a compiler, but not in an interpreter. By detecting this in the dynamically typed language Javascriptish, a lot of puzzles are reduced for the students.

The problems with scoping or the wrong number of parameters to a function are both detected at compile-time, or throw an error during
One advantage of detecting these kinds of problems with the ATA is that we can customise the error message, in relation to those messages given by the JavaScript interpreter.

The problem with not calling their declared function would not be detected by most compilers; however, there are a lot of Integrated Development Environments (IDE) that warn the students that they have not used their declared functions.

### Error messages

The task for the ATA is two-fold. The first is detecting errors, and the second is explaining the errors. The different errors that the ATA can find are given an ID, which can then be translated into an error message. This makes the error messages modular in such a way that we can provide them in different languages depending on the settings for the student. Furthermore, in the future, it would possible to give different error messages depending on the level of the student, and/or the age of the student. Since a teacher might explain things in different ways to children, teenagers, adults or pensioners it is suitable to let the ATA adapt its messages to the recipient. The different errors would, on the whole, be the same, but the feedback would be different. This is a strength in the implementation for the teacher since it will be possible for each teacher to customise the error messages for the students.

As stated in the section about assessment, the error messages should provide information to let the student figure out what went wrong and why in order to help them correct their code. This in comparison to compilers that poorly state what the error is (Traver, 2010).

An example of this is illustrated by the following error message (also shown in Figure 5.2).

> I could not find the variable myage in the current scope.  
> Remember that a variable declared inside a function is not available outside that function.

Here we declare *what* the error is (could not find the variable), and *why* it occurred (not declared where it is used).

This is also consistent with what Ramaprasad (1983) says about feedback, regarding that it is only feedback if the information can be used to alter the gap. In comparison with ordinary error messages, such as “Parse error on line X”, where the information could be argued only to contain information about the gap.
Types and parsability

In this implementation of the ATA, we have used static analysis to analyse the programs written by students to try to find common misconceptions about programming. In order to succeed with such an analysis, the program has to be parsable. That means that it has to be syntactically correct in order for the parser to understand it so that it can transform the program into an abstract syntax tree. Therefore, there are two steps for a complete ATA:

1. Help the student build a syntactically correct and parsable program
2. Help the student build a semantically correct program

More steps can be added, such as helping the student build a program that actually solves the given problem by comparing input data with expected output data or building programs that are efficient.

There are two different language paradigms when it comes to types: statically typed or dynamically typed. A language is statically typed if the type of a variable is known at the compile time. This can either be done by the programmer who specifies the type of each variable in the declaration (as in Java or C), or by type interference where the type system deduces the type of a variable (as in OCaml or Haskell). As a consequence, the compiler may check the types and therefore a lot of trivial bugs can be caught in the compile stage. With a dynamically typed language, the type is associated with runtime values and no static type-checking is done. This is a property of, for instance, scripting languages (as in Python or JavaScript). Some scripting languages have an interpreter that directly executes the program without having it compiled into machine code.

In this context, we use an untyped language. Thus, when the students write their programs, no type checking is done. One insight given by doing the implementation in the strongly typed language OCaml is the strength of the compiler. There have been many times where the program has not been compilable at first because of wrong types, but when the program finally compiles it also often works correctly. Therefore, by knowing and thinking of types, a novice might be helped in their progress of writing programs. On the other hand, those messages given by the OCaml compiler are difficult to understand and
might be difficult to accommodate for the student and therefore might be more suitable for a later course.

The constructivist theory of learning and formative assessment

According to the constructivist theory of learning students are creating their own schemas or models of the current topic. This may result in misconceptions and different persons probably will have different models even if the teacher is the same since it is an individual process. The thesis does not claim to identify the model held by the student, but rather focus on the gaps in the models and the things in the models that are not usable. An example would be that if a student thinks that variables will be accessible all over the program, the ATA would not focus on what the student knows about variables but instead focus on the pure misconception regarding the scope of the variables. The ATA does not know why the problem occurs in terms of why the student typed their code in a specific way. The ATA does know that the error occurred and focuses on giving feedback to that error in order to alter the students’ model.

Going back to the techniques used in formative assessment, the following three processes were identified:

- Establishing where the learners are in their learning.
- Establishing where they are going.
- Establishing what needs to be done to get them there.

with the following activities:

- Sharing success criteria with learners.
- Classroom questioning.
- Comment-only marking.
- Peer- and self-assessment.
- Formative use of summative tests.
The main activity utilised by the ATA is comment-only marking. Furthermore, by using the quizzes mentioned previously one could say that the ATA is using some kind of classroom questioning. This would also be the case when the ATA is collecting data of frequent error messages, to be able to get a picture of both the student’s progress as well as the progress of the whole course. The other three activities are not in the scope of the ATA at the moment but is something the course designer should include in a MOOC.

When it comes to the three processes, the establishment of where the students are going could be seen as the initial study of this thesis. The objective is to make students grasp the fundamentals of programming by avoiding the common misconceptions. Thus, we want the students to achieve a viable model without these misconceptions. Through detecting the misconception, the ATA is establishing where the student is in their learning process. By serving an error message that helps the student build a more correct model, one step is taken towards establishing what needs to be done to get them there. In addition, in the context of a MOOC, by routing the student towards more instructional films that describes the concepts the student has troubles with, the ATA is giving the students more tools to help them move forward in their learning process.

**Methodology discussion**

This thesis uses a phenomenographic study in order to identify common misconceptions about programming among novice programmers. As noted, this method is associated with description rather than analysis. Because of this, no attempt at understanding why these misconceptions occur is done. Instead, the results are handled as provided by the interviewed participants to build a foundation for the further work in the thesis.

When it comes to the reliability of the initial investigation, the results only reflect the perceptions of the interviewed teachers. This means that differences may occur depending on the interviewed group. If making the same study again, a larger group of teachers could be interviewed with a wider spread of experience and educational site in order to achieve higher reliability. In order to achieve a higher validity for the initial study, one could analyse old exams or observe students writing code at lab sessions. Nevertheless, it would be difficult to sim-
ulate similar learning contexts as when students write code on their own.

All errors identified in the initial study were inserted by the independent individuals, except for the problem with loops as mentioned. This makes it natural to question the significance of that particular misconception. At the same time, it is possible to argue that the results of the initial study along with the inductive approach give an evaluation result with a high validity.

The design of the error messages is not evaluated at all. This is something that can be done in order to achieve better feedback from the ATA. By interviewing teachers or students and measure their reactions to different error messages, one may conclude whether one message is better than the other. Something that could be done when the ATA is live is to do some kind of A/B-testing. That is a controlled experiment where some users get version A of an error message, and the rest get version B of an error message. Later, the conversion can be measured and the message with the best result can be considered as the one to use.
Chapter 7

Conclusion

This chapter concludes the thesis and discusses future research.

7.1 Future research

To be able to give a more versatile feedback to the student, more sorts of analysis should be added. As discussed earlier, this would include dynamic analysis. With added dynamic analysis, we would be able to test the correctness of programs in a new way, by comparing the expected output of the program with the provided output. This is an easy task when it comes to basic input-driven programs, but a completely different problem regarding graphics and games. How do we know that a click driven program is correct, such as an escape room game, or the classical pong game?

Another question that is a result of the work with this thesis is how heavily typed the taught language should be. Which are the benefits of teaching untyped languages compared to typed languages? Should the type be declared by the programmer, or deduced via type inference?

In order to effectively use the ATA in a future MOOC context, more focus should be given to the parser in the tool-chain as well. This since we want to help the student construct a parsable program first, in order to pass it further to the ATA. If the student does not manage to write parsable code, it will never reach the benefits of the ATA. Thus, the parser should be extended to give better feedback about the \textit{whats} and \textit{whys} of the parse errors as well.

Finally, more elements should be added to be able to work accord-
ing to more of the concepts in formative assessment. This includes quizzes and directing the student to instructions that go in line with their current knowledge.

7.2 Conclusion

The purpose of the thesis is to automate the interaction is usually performed between the student and the teacher or teaching assistant when assessing code, with focus on gaps in the students’ cognitive models. This is done with the constructivist perspective of knowledge and formative assessment as a theoretical framework.

This thesis aims to answer the question:

Which are the common misconceptions in programming for novice programming students?

in order to design a static analyser that detects common misconceptions about programming and provides feedback about what, why and where.

Teachers’ perceptions of common misconceptions about programming among novice programmers have been investigated in this thesis. That study identified 13 different misconceptions, and the following five misconceptions were the main focus for the rest of the study.

- Variables in different scopes.
- Usage of function parameters.
- Calling functions.
- Catching return values.
- If usage inside loops and breaking out of loops.

The results of that investigation laid the foundation for the design and implementation of the Artificial Teaching Assistant, which was evaluated by analysing program samples with inserted errors. The evaluation shows that the ATA detects the majority of the identified misconceptions and gives feedback about both what and why in accordance with theories about formative assessment. The ATA also manages to correctly position the errors in the majority of the cases. Hence, it is shown that it is possible to automate this interaction when it comes to the identified misconceptions.
Bibliography


Appendix A

Evaluation Programs

This appendix contains the source code for the working program examples as well as the modified ones.

A.1 Binary search

The binary search is a commonly taught algorithm in basic computer science courses. Listing A.1 shows the original code.

```javascript
/*
A program that uses binary search to find elements in an array of int
*/

function floor(num) {
    return num - (num % 1)
}

function binarySearch(items, value) {
    var startIndex = 0
    var stopIndex = items.length - 1
    var middle = floor((stopIndex + startIndex)/2)

    while(items[middle] != value && startIndex < stopIndex){
        print(middle)
        print(items[middle])
        //adjust search area
        if (value < items[middle]){    
            stopIndex = middle - 1
        } else {
            if (value > items[middle]){
```

```javascript
```
```
startIndex = middle + 1

// recalculate middle
middle = floor((stopIndex + startIndex)/2)
}
print(middle)

// make sure it’s the right value
if (items[middle] != value) {
    return -1
} else {
    return middle
}

// Test cases
var arr = [3,7,9,11,16,72,89]
if (binarySearch(arr,16) == 4) {
    print("Number 8 was found correctly")
} else {
    print("We could not find number 8 a place 4")
}

if (binarySearch(arr,57) == -1) {
    print("Number 57 does not exists in array, that is good")
} else {
    print("We were able to find number 57 in some strange way")
}
```

Listing A.1. The original binary search program
A.2 Bubble sort

Bubble sort is an easy-to-implement, but not that efficient sorting algorithm commonly taught in basic computer science courses. Listing A.2 shows the original code.

```javascript
/*
A program that uses bubble sort to sort one array of int and one array of string */

function bubbleSort(items) {
  var n = items.length
  var swapped = true
  while (swapped) {
    swapped = false
    var i = 1
    while (i < n) {
      if (items[i-1] > items[i]) {
        var tmp = items[i]
        items[i] = items[i-1]
        items[i-1] = tmp
        swapped = true
      }
      i = i + 1
    }
  }
  return items
}

function main() {
  var arr1 = [9,1,3,2,5,1,2,6,2]
  var arr2 = ["friday","cloud","morning","brownish","descends","every","A"]
  arr1 = bubbleSort(arr1)
  arr2 = bubbleSort(arr2)
  print(arr1)
  print(arr2)
}

main()
```

Listing A.2. Bubble sort program
A.3 Find the smallest number divisible by a range

This is a program that finds the smallest number divisible by a range of numbers. For instance is the smallest number divisible by the interval [1, 3] 6, and the smallest number divisible by [1, 5] is 60.

```javascript
// Find the smallest number that is divisible by the numbers 1 to limit
// https://projecteuler.net/problem=5

class is_divisible_up_to(number, limit) {
    var i = 2
    while (i <= limit) {
        if (number % i != 0) {
            return false
        }
        i = i + 1
    }
    return true
}

function find_smallest_number(limit) {
    var number = limit*limit
    while (true) {
        if (is_divisible_up_to(number, limit)) {
            return number
        }
        number = number + 1
    }
}

print(find_smallest_number(20))
```

Listing A.3. Program to find smallest number divisible by range
A.4 List primes

Another quite common problem is to find all primes up to a certain number. This is an exercise that primarily trains loops and functions.

```javascript
// Program that lists all primes up to number n

function is_prime(number) {
    if (number <= 1) {
        return false
    }
    var i = 2
    while (i*i < number) {
        if (number % i == 0) {
            return false
        }
        i = i + 1
    }
    return true
}

function find_primes(limit) {
    var i = 2
    while (i < limit) {
        if (is_prime(i)) {
            print(i)
        }
        i = i + 1
    }
}

find_primes(1000)
```

Listing A.4. Program to list all primes up to number \textit{limit}
A.5 Polynomial division

As a longer program example, a program for dividing polynomials is provided.

```javascript
function pop(arr) {
    var n = arr.length
    var i = 0
    var newarr = []
    while ( i < n - 1) {
        newarr[i] = arr[i]
        i = i + 1
    }
    return newarr
}

function copy(arr) {
    var n = arr.length
    var i = 0
    var newarr = []
    while ( i < n ) {
        newarr[i] = arr[i]
        i = i + 1
    }
    return newarr
}

function degree(poly) {
    while ( poly[poly.length-1] == 0) {
        poly = pop(poly)
    }
    return poly.length - 1
}

function zeros(num) {
    var i = 0
    var arr = []
    while ( i < num ) {
        arr[i] = 0
        i = i + 1
    }
    return arr
}

function polyShiftRight(p, places) {
    if ( places <= 0 ) {
        return p
    }
    // Rest of the function
}
```javascript
var pd = degree(p)
var d = copy(p)
var i = pd
while ( i >= 0 ) {
    d[i+places] = d[i]
    d[i] = 0
    i = i - 1
}
return d

function polyMultiply(p, m) {
    var i = 0
    var n = p.length
    while ( i < n ) {
        p[i] = p[i] * m
        i = i + 1
    }
}

function polySubtract(p, s) {
    var i = 0
    var n = p.length
    while ( i < n ) {
        p[i] = p[i] - s[i]
        i = i + 1
    }
}

function poly_print(poly) {
    var pd = degree(poly)
    var i = pd
    var result = ""
    while ( i >= 0 ) {
        var coeff = poly[i]
        if ( coeff != 0 ) {
            if ( coeff == 1 ) {
                if ( i < pd ) {
                    result = result + " + "
                }
            } else {
                if ( coeff == -1 ) {
                    if ( i < pd ) {
                        result = result + " - "
                    } else {
                        result = result + "-"
                    }
                }
                result = result + " + "
            }
        }
        i = i - 1
    }
    return result
}
```

else {
    if ( coeff < 0 ) {
        if ( i < pd ) {
            result = result + " - " + -coeff
        } else {
            result = result + coeff
        }
    } else {
        if ( i < pd ) {
            result = result + " + " + -coeff
        } else {
            result = result + coeff
        }
    }
}

if ( i > 1 ) {
    result = result + "x^" + i
} else {
    if ( i == 1 ) {
        result = result + "x"
    }
}

i = i - 1

print(result)

function poly_div(N, D) {
    var dd = degree(D)
    var nd = degree(N)
    if ( dd < 0 ) {
        print("DIVISION BY ZERO")
    } else {
        var n2 = copy(N)
        var q = zeros(nd)
        while ( nd >= dd ) {
            var d2 = polyShiftRight(D, nd-dd)
            q[nd - dd] = n2[nd] / d2[nd]
            polyMultiply(d2, q[nd-dd])
            polySubtract(n2, d2)
            nd = degree(n2)
        }
        print("Quotient:")
        poly_print(q)
print("Reminder:")
poly_print(n2)
}
}

function main() {
    var n = [-42, 0, -12, 1]
    var d = [-3, 1, 0, 0]
    print("Numerator: ")
poly_print(n)
    print("Denominator: ")
poly_print(d)
poly_div(n, d)
}

main()

Listing A.5. Program for polynomial division
Appendix B

Source code of the ATA

```ml
open Printf
open Ast
open Ustring.Op
open Msg
(*
Javascriptish is licensed under the MIT license.
Copyright (C) David Broman. See file LICENSE
This file is created by Johan Myrsmeden.
*)

The main file for performing program analysis.

module StringMap = Map.Make (String)

type environment_info =
  | ErrorMsg of message (* See msg.ml *)
  | VariableInfo of info * ustring (* fi, name *)
  | FunctionInfo of info * ustring * int * bool * bool * ustring list (* name, number_of_arguments, called?, non_void?, parameter names *)

let print_env_info item =
  match item with
  | VariableInfo(_, name) -> uprint_string(us"Variable: " ^_. name); print_string "\n"
  | ErrorMsg(msg) -> uprint_endline (message2str(msg))
  | FunctionInfo (_,name,_,_,_,_) -> uprint_string(us"Function: " ^_. name); print_string "\n"

(* available_functions is used so the ATA knows which functions to have as suggestions in spell checking *)
```
```ocaml
let available_functions = [FunctionInfo(NoInfo, us"print", 0, false, false, [])]

(* Function to append two lists *)
let rec loop acc l1 l2 = 
  match l1, l2 with 
  | [], [] -> List.rev acc 
  | [], h :: t -> loop (h :: acc) [] t 
  | h :: t, l -> loop (h :: acc) t l 
  in 
  loop [] l1 l2 

(* Function to print a list of different environment_infos *)
let rec print_list lst = 
  let rec loop lst = 
    match lst with 
    | [] -> () 
    | x::xs -> print_env_info x; loop xs 
  in loop lst 

(* A reduce function : apply function f to each element in list 
and collect in accumulator *)
let rec reduce f lst acc = 
  match lst with 
  | [] -> acc 
  | x::xs -> append(f x acc) (reduce f xs acc) 

(* A function to loop over elements in lst 
calling function f with accumulator acc *)
let rec loop f lst acc = 
  match lst with 
  | [] -> acc 
  | x::xs -> loop f xs (f x acc) 

let rec boolean_reduce f lst = 
  match lst with 
  | [] -> false 
  | x::xs -> (f x) || (boolean_reduce f xs) 

(* Does name exists in lst? *)
let rec exists name lst = 
  match lst with 
  | [] -> false 
  | x::xs -> (match x with 
  | VariableInfo(_,name2) -> if Ustring.equal name name2 
```

The code snippet above defines several functions. Here's a brief description of each:

1. `available_functions`: A list of function information, where each function is represented by a `FunctionInfo` object.
2. `append`: A recursive function that appends two lists together.
3. `print_list`: A function that prints a list of different environmentInfos.
4. `reduce`: A function that applies a given function `f` to each element in a list, accumulating results.
5. `loop`: A function that loops over a list, applying a given function `f` to each element and accumulating results.
6. `boolean_reduce`: A function that reduces a list using a boolean function `f`, returning `true` if any element satisfies the condition.
7. `exists`: A function that checks if a given name exists in a list of environmentInfos.

These functions are designed to work with lists and environmentInfos, providing tools for appending lists, printing lists, reducing lists using functions, and checking for the existence of names within lists.
then true else (exists name xs)
| FunctionInfo(_,name2,_,_,_,_,_) -> if Ustring.equal
  name name2 then true else (exists name xs)
| _ -> false
)
75
76 (* Check if item exists in env-part of map env *)
77 let exists_in_environment name env lst_name =
  exists name (StringMap.find lst_name env)
79
80 (* Functions to handle environment
  Environment is a map with lists of environment_info, with
  string keys *)
82 let get_empty_environment lst_names =
84     match lst with
  | [] -> StringMap.empty
  | x::xs -> StringMap.add x [] (loop xs)
87 in loop lst_names
88
89 let print_errors lst headline =
  print_endline headline;
90 print_list lst
92
94 (* Function that keeps the lists from old_env given by the
  keep_lst and replaces the one given by replace *)
95 let get_scope_environment env old_env keep_lst replace =
97     let rec loop lst =
  | [] -> StringMap.empty
  | x::xs -> StringMap.add x (StringMap.find x env) (loop xs)
100 in
101 let new_map = loop keep_lst in
102 let old_content = StringMap.find replace old_env in
103 StringMap.add replace old_content new_map
106
108 (* Function to add variable to environment *)
109 let add_env_var env lst_name var =
  let current = StringMap.find lst_name env in
111 let new_content = var::current in
113 StringMap.add lst_name new_content env
115
117 (* Mark function with provided name as called *)
118 let mark_as_called env function_name =
120     match lst with
| [] -> []
| x::xs -> match x with
| FunctionInfo(fi, name, num_args, called, non_void, params) -> (  
  if Ustring.equal name function_name then
    let new_function = FunctionInfo(fi, name, num_args, true, non_void, params) in
    new_function::xs
  else
    x::(loop xs)
  )
| _ -> loop xs
in StringMap.add "function_definitions" (loop (StringMap.find "function_definitions" env)) env

(* Check if a function does return a value *)
let does_return_value env function_name =
let rec loop lst =
  match lst with
  | [] -> false
  | x::xs -> match x with
    | FunctionInfo(_,name,_,_,non_void,_) -> (  
      if Ustring.equal name function_name then
        non_void
      else
        loop xs
    )
  | _ -> loop xs
in loop (StringMap.find "function_definitions" env)

(* Get the number of parameters for the function with the provided name in a list of environment infos (variant function info) *)
let get_num_params_in_list lst name =
let rec loop lst =
  match lst with
  | [] -> 0
  | x::xs -> (  
    match x with
      | FunctionInfo(info,func_name,num_params,_,_,_) -> if
        Ustring.equal name func_name then num_params else
        (loop xs)
      | _ -> loop xs
  )
in loop lst

(* Get the parameters for the function with the provided name in a list of environment infos (variant function info) *)
APPENDIX B. SOURCE CODE OF THE ATA

```ml

let get_params_in_list lst name =
  let rec loop lst =
    match lst with
      | [] -> []
      | x::xs -> (
        match x with
          | FunctionInfo(info,func_name,_,_,_,params) -> if
            Ustring.equal func_name name then params else (      
              loop xs)
          | _ -> loop xs)
      in loop lst

(* Get minimum value of a, b and c *)
let minimum a b c =
  if a < b then (      
    if a < c then
      a
    else
      c
  )
else (      
  if b < c then
    b
  else
    c
)

(* The Levenshtein Distance algorithm. Gives the lowest number of mutations between Ustring s and Ustring t *)
let rec levenshtein_distance s len_s t len_t =
  if len_s = 0 then len_t
  else (      
    if len_t = 0 then len_s
    else (      
      let cost = if Ustring.equal (Ustring.sub s (len_s-1) 1)
        (Ustring.sub t (len_t-1) 1) then
        0
      else
        1 in
      minimum
        ((levenshtein_distance s (len_s-1) t len_t) + 1)
        ((levenshtein_distance s len_s t (len_t-1)) + 1)
        ((levenshtein_distance s (len_s-1) t (len_t-1)) +
          cost)
    )
  )

)
(* Function to find possible names if the provided name is misspelled *)

let find_possible_name name env lst_name =
  let rec loop lst distance suggestion =
    match lst with
    | [] -> suggestion
    | x::xs -> (match x with
              | VariableInfo(_, x) -> (let dist =
                                      levenshtein_distance name (Ustring.length name)
                                      x (Ustring.length x) in
                                      if dist < distance then
                                        (loop xs dist x)
                                      else
                                        (loop xs distance suggestion))
              | FunctionInfo(_,x,_,_,_,_) -> (let dist =
                                                   levenshtein_distance name (Ustring.length name)
                                                   x (Ustring.length x) in
                                                   if dist < distance then
                                                     (loop xs dist x)
                                                   else
                                                     (loop xs distance suggestion))
              | ErrorMsg(_) -> us"
    in if String.equal lst_name "functions" then
       loop (append (StringMap.find "function_definitions" env) available_functions) 5 (us"
    else loop (StringMap.find lst_name env) 5 (us"

(* Function to check if the term ends with a return statement or not *)

let rec is_non_void tm =
  match tm with
  | TmDef(fi,isconst,name,tm) -> is_non_void tm
  | TmWhile (fi, tm_head, tm_body) -> is_non_void tm_body
  | TmIf(fi,tm1,tm2,tm3) -> is_non_void tm2 || (match tm3 with
     | Some(tm) -> is_non_void tm
     | None -> false )
  | TmAssign(fi,name,tm) -> is_non_void tm
  | TmRet(fi,tm) -> true
  | TmConst(fi,const) -> false
| TmFunc(fi, params, tm) -> is_non_void tm
| TmCall(fi, tm, tmlist) -> false
(* Other *)
| TmScope(fi, tmlist) -> boolean_reduce is_non_void tmlist
| TmBreak(fi) -> false
| TmProj(fi, tm, name) -> false
| TmArrIndex(fi, name, index) -> false
| TmArray(fi, items) -> false

(* Check if a function returns a value and if it is caught or not *)

let check_function_for_return env tm in_assignment =
match tm with
| TmVar(fi2, isconst2, name) -> {
  if not in_assignment && does_return_value env name then
    let error = ErrorMsg(UNCAUGHT_RETURN, WARNING, fi2, [name]) in
    add_env_var env "errors" error
  else
    env
| _ -> env

(* The function that handles calls, and checks for *)
(* Returns *)
(* Number of parameters *)
(* Argument definitions *)

let rec handle_tm_call f env tm tmlist in_assignment =
match tm with
| TmVar(fi2, isconst2, name) -> {
  loop (fun tm2 acc ->
    (match tm2 with
      | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
      | _ -> f tm2 acc)) tmlist (
      (* Check if this function is misspelled *)
      if not (exists name (append (StringMap.find "function_definitions" env) available_functions))
      then
        (let suggestion = find_possible_name name env "functions" in
        let error = ErrorMsg(FUNCTION_NOT_FOUND, ERROR, fi2 , [name;suggestion]) in
        add_env_var env "errors" error)
      else
        (let env = check_function_for_return env tm in_environment in
        if exists_in_environment name env "


function_definitions" then (* Since this is a function call, we want to check if it is a defined function (in comparison to for instance print) *)

(* Mark function as called *)

let env = mark_as_called env name in (* Calculate number of parameters and arguments *)

let expected_num_params = get_num_params_in_list (StringMap.find "function_definitions" env) name in

let provided_num_args = List.length tmlist in
if provided_num_args <> expected_num_params then

let error = ErrorMsg(WRONG_NUMBER_OF_PARAMS, ERROR, fi2, [name; (ustring_of_int expected_num_params); (ustring_of_int provided_num_args)]) in add_env_var env "errors" error

else env

else

env)))

| TmConst(fi, const) -> (* We are using a const function, which means that we are handling return value *)

loop (fun tm2 acc -> 
(match tm2 with
 | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
 | _ -> f tm2 acc)
 | _ -> loop f tmlist env)

(* Function to analyze scope. env is a StringMap accumulator *)

let analyze_scope ast errors =

let rec traverse ast env =

match ast with

(* Statements *)

| TmDef(fi, isconst, name, tm) ->

(match tm with
 | TmFunc(fi2, params, tm2) -> let data = FunctionInfo (fi, name, (List.length params), false, (is_non_void tm), params) in traverse tm (add_env_var env "function_definitions" data)
 | TmVar(fi2, isconst2, name2) -> let varinfo = VariableInfo (fi, name) in traverse tm (add_env_var env "env" varinfo)
 | TmAssign(fi2, tmleft, tm2) ->

| TmConst(fi, const) -> (* We are using a const function, which means that we are handling return value *)

loop (fun tm2 acc -> 
(match tm2 with
 | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
 | _ -> f tm2 acc)
 | _ -> loop f tmlist env)

(* Function to analyze scope. env is a StringMap accumulator *)

let analyze_scope ast errors =

let rec traverse ast env =

match ast with

(* Statements *)

| TmDef(fi, isconst, name, tm) ->

(match tm with
 | TmFunc(fi2, params, tm2) -> let data = FunctionInfo (fi, name, (List.length params), false, (is_non_void tm), params) in traverse tm (add_env_var env "function_definitions" data)
 | TmVar(fi2, isconst2, name2) -> let varinfo = VariableInfo (fi, name) in traverse tm (add_env_var env "env" varinfo)
 | TmAssign(fi2, tmleft, tm2) ->

| TmConst(fi, const) -> (* We are using a const function, which means that we are handling return value *)

loop (fun tm2 acc -> 
(match tm2 with
 | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
 | _ -> f tm2 acc)
 | _ -> loop f tmlist env)

(* Function to analyze scope. env is a StringMap accumulator *)

let analyze_scope ast errors =

let rec traverse ast env =

match ast with

(* Statements *)

| TmDef(fi, isconst, name, tm) ->

(match tm with
 | TmFunc(fi2, params, tm2) -> let data = FunctionInfo (fi, name, (List.length params), false, (is_non_void tm), params) in traverse tm (add_env_var env "function_definitions" data)
 | TmVar(fi2, isconst2, name2) -> let varinfo = VariableInfo (fi, name) in traverse tm (add_env_var env "env" varinfo)
 | TmAssign(fi2, tmleft, tm2) ->

| TmConst(fi, const) -> (* We are using a const function, which means that we are handling return value *)

loop (fun tm2 acc -> 
(match tm2 with
 | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
 | _ -> f tm2 acc)
 | _ -> loop f tmlist env)

(* Function to analyze scope. env is a StringMap accumulator *)

let analyze_scope ast errors =

let rec traverse ast env =

match ast with

(* Statements *)

| TmDef(fi, isconst, name, tm) ->

(match tm with
 | TmFunc(fi2, params, tm2) -> let data = FunctionInfo (fi, name, (List.length params), false, (is_non_void tm), params) in traverse tm (add_env_var env "function_definitions" data)
 | TmVar(fi2, isconst2, name2) -> let varinfo = VariableInfo (fi, name) in traverse tm (add_env_var env "env" varinfo)
 | TmAssign(fi2, tmleft, tm2) ->

| TmConst(fi, const) -> (* We are using a const function, which means that we are handling return value *)

loop (fun tm2 acc -> 
(match tm2 with
 | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
 | _ -> f tm2 acc)
 | _ -> loop f tmlist env)

(* Function to analyze scope. env is a StringMap accumulator *)

let analyze_scope ast errors =

let rec traverse ast env =

match ast with

(* Statements *)

| TmDef(fi, isconst, name, tm) ->

(match tm with
 | TmFunc(fi2, params, tm2) -> let data = FunctionInfo (fi, name, (List.length params), false, (is_non_void tm), params) in traverse tm (add_env_var env "function_definitions" data)
 | TmVar(fi2, isconst2, name2) -> let varinfo = VariableInfo (fi, name) in traverse tm (add_env_var env "env" varinfo)
 | TmAssign(fi2, tmleft, tm2) ->

| TmConst(fi, const) -> (* We are using a const function, which means that we are handling return value *)

loop (fun tm2 acc -> 
(match tm2 with
 | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
 | _ -> f tm2 acc)
 | _ -> loop f tmlist env)

(* Function to analyze scope. env is a StringMap accumulator *)

let analyze_scope ast errors =

let rec traverse ast env =

match ast with

(* Statements *)

| TmDef(fi, isconst, name, tm) ->

(match tm with
 | TmFunc(fi2, params, tm2) -> let data = FunctionInfo (fi, name, (List.length params), false, (is_non_void tm), params) in traverse tm (add_env_var env "function_definitions" data)
 | TmVar(fi2, isconst2, name2) -> let varinfo = VariableInfo (fi, name) in traverse tm (add_env_var env "env" varinfo)
 | TmAssign(fi2, tmleft, tm2) ->

| TmConst(fi, const) -> (* We are using a const function, which means that we are handling return value *)

loop (fun tm2 acc -> 
(match tm2 with
 | TmCall(fi3, tm3, tmlist3) -> handle_tm_call f acc tm3 tmlist3 true
 | _ -> f tm2 acc)
 | _ -> loop f tmlist env)
(match tmleft with
  | TmVar(fi3, isconst3, name) -> let varinfo = VariableInfo(fi, name) in traverse tm (add_env_var env "env" varinfo)
  | _ -> traverse tm env)

| TmArray(fi3, items) -> let varinfo = VariableInfo(fi, name) in traverse tm (add_env_var var (loop
  (fun item acc -> (match item with
    | TmVar(fi4, isconst, name) ->
      (if exists_in_environment name env "env" then
        acc
      else
        (let suggestion = find_possible_name name env
         "env" in
          let error = ErrorMsg(VAR_NOT_IN_SCOPE, ERROR, fi4, [name;suggestion]) in
          add_env_var acc "errors" error))

  | TmConst(fi2, const2) -> let varinfo = VariableInfo(fi, name) in traverse tm (if isconst then env else
    (add_env_var env "env" varinfo))

  | TmCall(fi2, tm2, tmlist) -> (* A call in a definition means that we handle a return value *)
    let varinfo = VariableInfo(fi, name) in
    handle_tm_call traverse (add_env_var env "env" varinfo) tm2 tmlist true

| _ -> let varinfo = VariableInfo(fi, name) in
trace tm (add_env_var env "env" varinfo))

| TmWhile (fi, tm_head, tm_body) -> traverse tm_body (match tm_head with
  | TmCall(fi1, tm, tmlist) -> handle_tm_call traverse
    env tm tmlist true

| _ -> traverse tm_head env)

| TmIf(fi, tm1, tm2, tm3) -> (
  let env = traverse tm2 (match tm3 with
    | Some(tm) -> traverse tm env

    | None -> env ) in

  match tml with
  | TmCall(fi1, tm1, tmlist) -> handle_tm_call traverse
    env tm1 tmlist true

| _ -> traverse tm1 env)

| TmAssign(fi, tmleft, tmright) ->

(match tmleft with
| TmVar(fi2, isconst2, name) ->
(if exists_in_environment name env "env" then
  (match tmright with
    | TmCall(fi2, tm2, tmlist) -> handle_tm_call
      traverse env tm2 tmlist true (* Handling return value *)
    | _ -> traverse tmright env)
  else
    (let suggestion = find_possible_name name env "env" in
     let error = ErrorMsg(VAR_NOT_IN_SCOPE, ERROR, fi,
      [name; suggestion]) in
      let env = add_env_var env "errors" error in
      match tmright with
    | TmCall(fi2, tm2, tmlist) -> handle_tm_call
      traverse env tm2 tmlist true (* Handling return value *)
    | _ -> traverse tmright env))

| TmArrIndex(fi2, var, index) ->
  (match var with
    | TmVar(fi2, isconst2, name) ->
      (if exists_in_environment name env "env" then
        (match tmright with
          | TmCall(fi2, tm2, tmlist) -> handle_tm_call
            traverse env tm2 tmlist true (* Handling return value *)
          | _ -> traverse tmright env)
        else
          (let suggestion = find_possible_name name env "env" in
           let error = ErrorMsg(VAR_NOT_IN_SCOPE, ERROR, fi,
            [name; suggestion]) in
            let env = add_env_var env "errors" error in
            match tmright with
          | TmCall(fi2, tm2, tmlist) -> handle_tm_call
            traverse env tm2 tmlist true (* Handling return value *)
          | _ -> traverse tmright env))
      | _ -> traverse tmright env)
    | _ -> traverse tmright env)
    | TmRet(fi, tm) -> (match tm with
      | TmCall(fi2, tm2, tmlist) -> handle_tm_call traverse
        env tm2 tmlist true (* Handling return value *)
      | _ -> traverse tm env)
      (* Expressions *)
    | TmVar(fi, isconst, name) ->
      if exists_in_environment name env "env" then
      (* Expressions *)
    | _ -> traverse tm right env)
(let suggestion = find_possible_name name env "env" in
let error = ErrorMsg(VAR_NOT_IN_SCOPE, ERROR, fi, [name;suggestion]) in
add_env_var env "errors" error)

| TmConst(fi,const) -> env
| TmFunc(fi,params,tm) -> traverse tm (loop (fun name
   acc -> let varinfo = VariableInfo(fi, name) in
   add_env_var acc "env" varinfo) params env)
| TmCall(fi,tm,tmlist) -> (* If we are here, we are not in an assignment *)
handle_tm_call traverse env tm tmlist false
| TmBreak(fi) -> env
| TmProj(fi,tm,name) -> env
| TmArrIndex(fi,tm,index) -> env
| TmArray(fi,items) -> env
(+ Other *)
| TmScope(fi,tmlist) -> get_scope_environment (loop
   traverse tmlist env) env ["errors"; "function_definitions"] "env"

let environment = traverse ast (StringMap.add "errors"
errors (get_empty_environment ["env"; "function_definitions"])) in

let errors = StringMap.find "errors" environment in
let function_definitions = StringMap.find "function_definitions" environment in
loop (fun x acc ->
  | FunctionInfo(fi,name,num_params,called,non_void,_) ->
    (if not called then
      let error = ErrorMsg(FUNCTION_NOT_CALLED, WARNING,
        fi, [name]) in
      error::acc
    else
      acc)
    | _ -> acc) function_definitions errors

(* Function that checks for patterns
that indicate the usage of flag variables instead of break statements *)
let check_loops ast errors =
let rec traverse ast env =
  match ast with
/* Statements */
| TmDef(fi,isconst,name,tm) -> traverse tm env
| TmWhile (fi, tm_head, tm_body) -> (match tm_body with
|   | TmScope(fi, tmlist) ->
|   |   (loop (fun tm env ->
|   |     (match tm with
|   |       | TmIf(fi,tm1,tm2,tm3) ->
|   |         |   (match tm2 with
|   |         |       | TmScope(fi, tmlist) ->
|   |         |       |   (loop (fun tm5 env ->
|   |         |       |     (match tm5 with
|   |         |       |       | TmAssign(fi, tmleft, tm) -> (match
|   |         |       |       |     tmleft with
|   |         |       |       |       | TmVar(fi1, isconst2, name) ->
|   |         |       |       |       |   let varinfo = VariableInfo(fi, name) in
|   |         |       |       |       |     add_env_var env "booleans" varinfo
|   |       | _ -> env)
|   |       | _ -> env)
|   |  ) tmlist env)
|   | _ -> env)
|   | _ -> env)
|   | TmIf(fi,tm1,tm2,tm3) ->
|   |   (match tm1 with
|   |       | TmVar(fi2, isconst, name) -> (if exists_in_environment name env "booleans" then
|   |         |     let error = ErrorMsg(BOOLEAN_INSTEAD_OF_BREAK, WARNING, fi2, []) in
|   |         |     add_env_var env "errors" error
|   |         |   else
|   |         |     env)
|   |         | _ -> env)
|   |         | TmAssign(fi,tmleft,tm) ->
|   |         |   (match tmleft with
|   |         |       | TmVar(fi2, isconst2, name) -> (let varinfo = VariableInfo(fi, name) in
|   |         |         | add_env_var env "env" varinfo)
|   |         |       | _ -> env)
|   |         | TmRet(fi,tm) -> env
| * Expressions *
| TmVar(fi,isconst,name) -> env
| TmConst(fi,const) -> env
| TmFunc(fi,params,tm) -> traverse tm (loop (fun name
acc -> let varinfo = VariableInfo(fi, name) in
add_env_var acc "env" varinfo) params env

| TmCall(fi,tm,tmlist) -> env
| TmBreak(fi) -> env
| TmProj(_,_,_) -> env
| TmArrIndex(_,_,_) -> env
| TmArray(_,_) -> env
(* Other *)
| TmScope(fi,tmlist) -> loop traverse tmlist env

let environment = traverse ast (StringMap.add "errors" errors (get_empty_environment ["env";"booleans"])) in

StringMap.find "errors" environment

(* Our main function, called from jsh.ml when program is ran with argument 'analyze' *)

let analyze ast =

let analyze_results = analyze_scope ast [] in

let loop_results = check_loops ast analyze_results in

if (List.length loop_results) > 0 then
  print_errors loop_results "Found errors:"