UNIVERSITY
OF OSLO

Master's thesis

Investigating Intermediate Student Transfer to Functional Programming

Jørgen Spilling

Informatics: Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Jørgen Spilling

Investigating Intermediate Student Transfer to Functional Programming

Supervisors:
Quintin Cutts
Ragnhild Kobro Runde

Abstract

The aim of this thesis is to validate the Model for Programming Language Transfer (MPLT) for the transfer from Object-Oriented to Functional programming, as well as to explore the views of students and lecturers regarding the use of programming language transfer to learn and teach programming languages, more specifically functional programming languages, by employing previously learned knowledge to facilitate acquiring new knowledge.
Taking inspiration from linguistics theories of transfer to a second natural language, the validated MPLT was developed by Dr. Ethel Thsukudu[1] to predict CS students' experience transferring from a previous programming language to a new one. As previous studies of this model have focused on novice students transferring between languages with relatively similar syntax and paradigms, this study will instead use the MPLT to focus on intermediate students with Object-Oriented backgrounds making the leap to Functional Programming.
The thesis claim is that: *The MPLT transfer categories are in evidence when intermediate students from an object-oriented background transfer to functional programming, and exploring this transfer with students and their teachers could improve the process.* Investigating students at both the University of Glasgow and the University of Oslo, the study takes a close look at students proficient in at least Java and Python as they begin learning Haskell and Scheme. Data was gathered using both a guess quiz administered to students at the start of their semesters, as well as interviews with students, teaching assistants and lecturers. The findings show that the MPLT transfer categories are in evidence when intermediate students used to object-orientation learn functional programming, and demonstrate that exploring this transfer can definitely improve this process. Among other interesting results, it was found that mutation has a serious, negative impact on students' ability to transfer from Object-Oriented to Functional Programming.

Keywords:

Transfer, MPLT, syntax, semantics, Scheme, Haskell, TCC, FCC, ATCC.

Acknowledgments I would like to extend my deepest thanks to everyone who has helped me on my journey to make this thesis a reality.
To my two amazing supervisors: I want to thank Quintin Cutts for his great wit, his costructive feedback as well as the patience to keep constantly challenging my ideas, providing the whetstone against which I've sharpened all the points I try to make. I also thank Ragnhild Kobro Runde, for her encouragment and calming presence, for holding me accountable to regulations and guidelines, for pointing me in the right direction and ensuring I make it to the goal post. I'm ever grateful to the both of you for your support and guidance, it has been my honour to learn from you both.
Along the way I've worked side-by-side with fellow master student Lars Kristian Skaarseth. I loved our chats, and it was great to have someone who could relate to my struggles. You were an inspiration to me, and I look forward to working with you in the future.
I am very grateful to everyone who participated in my studies for this project, who've worked so hard on my behalf to help me in my investigations, and I hope my findings can in some small way help return the favour.
I'd like to thank Cybernetisk Selskab, the student organization who took me into their fold and provided me with a sense of community and fellowship, as well as many hours of fun throughout all my student years.
Thanks to my parents, both for raising me and for providing advice and comfort whenever I've needed it.
Last, but not least, I thank my partner Lenka, for being an endless wellspring of love and patience for me throughout every challenge I've had to face in these last years.

Contents

1 Introduction **7**

1.1 Hypothesis . . . 8

1.2 Research questions . . . 8

1.2.1 What outcomes would be predicted by the MPLT for an intermediate student's first encounter with FP . . . 8

1.2.2 How does the intermediate students' experience transferring to FP match the predictions of the MPLT? . . . 8

1.2.3 What is the effect of students with prior programming transfer experience on the predictions, effectiveness and usefulness of the MPLT by comparison with novices transferring? . . . 8

1.2.4 How does having an OO background impact students' experience transferring to FP? . . . 8

1.2.5 What do teachers think about transfer and what is their current practice? . . . 9

1.2.6 Are there issues of transfer to FP beyond the MPLT? . . . 9

1.2.7 How could exploring transfer aid in teaching FP? . . . 9

1.3 Study setting . . . 9

1.4 Thesis structure . . . 9

2 Background **11**

2.1 Literature review . . . 11

2.1.1 Novices learning Functional Programming . . . 11

2.1.2 Expert programmers learning new Lanugages . . . 12

2.1.3 Intermediate student programmers . . . 13

2.1.4 PL knowledge transfer . . . 13

2.1.5 Model for Programming Language Transfer . . . 14

2.1.6 Summary of literature review . . . 15

2.2 Programming languages . . . 16

2.2.1 Object-Oriented Programming Languages . . . 16

2.2.2 Functional Programming Languages . . . 17

3 Methodology **19**

3.1 Guess quiz . . . 19

3.1.1 Glasgow guess quiz . . . 19

3.1.2 UIO guess quiz . . . 20

3.1.3 UIO novice programmers guess quiz . . . 20

3.1.4 Analysis . . . 20

3.2 Guess quiz difficulties . . . 21

3.2.1 Hard-to-test concepts . . . 21

3.2.2 The problem with non-answers 22
3.3 Student interviews 22
3.3.1 Participants 22
3.3.2 Interview structure 22
3.4 Interviews with Teacher assistants 23
3.4.1 Reason for interviewing TAs 23
3.4.2 Interview structure 23
3.5 Interviews with lecturers 24
3.5.1 Reason for interviewing lecturers 24
3.5.2 Interview structure 24
3.6 Thematic analysis 24

4 Predicting transfer 27
4.1 How transfer is predicted 27
4.1.1 Transfer categories 27
4.1.2 Predicting transfer 27
4.2 Predicting transfer from Java/Python to Scheme 28
4.2.1 TCC - Logical operators: *and* and *or* 29
4.2.2 FCC - If-expression 30
4.2.3 ATCC - Cond-expression 31
4.3 Predicting transfer to Haskell 31
4.3.1 TCC: String 32
4.3.2 FCC: Function-definition 33
4.3.3 ATCC: Let-expression and local variables 34

5 Constructing the Guess Quizzes 35
5.1 UIO Guess Quiz 35
5.2 Glasgow guess quiz 36

6 Results 39
6.1 Guess quiz 39
6.1.1 Haskell guess quiz 39
6.1.2 Scheme guess quiz 41
6.1.3 Novice students Scheme guess quiz 43
6.1.4 Lessons learned 44
6.2 Interviews 45
6.2.1 Student interview analysis 45
6.2.2 TA interview analysis 49
6.2.3 Lecturer analysis 51

7 Discussion 59
7.1 RQ1: What outcomes would be predicted by the MPLT for an intermediate student's first encounter with FP 59
7.2 RQ2: How does the intermediate students' experience transferring to FP match the predictions of the MPLT 59
7.2.1 Accuracy of predictions 59
7.2.2 Students fixate on syntax 61
7.2.3 RQ2 discussion summary 61

7.3 RQ3: What is the effect of students with prior programming transfer experience on the predictions/effectiveness/usefulness of the MPLT by comparison with novices transferring? 62
7.3.1 Intermediate students' transfer compared to novice student transfer 62
7.3.2 RQ3 discussion summary 63
7.4 RQ4: How does having an OO background impact students' experience transferring to FP? 63
7.4.1 Interference of mutation on FP transfer 64
7.4.2 Students prefer loops over recursion 65
7.4.3 RQ4 discussion summary 65
7.5 RQ5: What do teachers think about transfer and what is their current practice? 66
7.5.1 Lecturers' thoughts on transfer 66
7.5.2 Lecturers' use of transfer 67
7.5.3 Teaching for transfer requires knowledge of students' backgrounds 67
7.5.4 RQ5 discussion summary 68
7.6 RQ6: Are there issues of transfer beyond the MPLT? 68
7.6.1 Paradigms 68
7.6.2 Abstract concepts 69
7.6.3 RQ6 discussion summary 69
7.7 RQ7: How could exploring transfer aid in teaching FP? 69
7.7.1 Teaching TCCs, FCCs and ATCCs 70
7.7.2 Awareness of what students transfer will help lecturers teach FP 70
7.7.3 RQ7 discussion summary 71

8 Future research 73
8.1 Compare transfer to different FP languages 73
8.2 Further exploring the influence of mutation on learning FP 73

9 Conclusion 75

A Guess quizzes 79
A.1 Haskell guess quiz 79
A.2 Scheme guess quiz 80

B Interview guides 85
B.1 Interview guide students 85
B.2 Interview guide Teaching Assistants 90
B.3 Interview guide IFI lecturers 95
B.4 Interview guide Glasgow lecturers 98

C Consent form 101

Chapter 1

Introduction

In an ever growing world of computer science and programming languages, it is imperative for any programmer to keep up with the times. Whether it is to remain employable, to improve a business, to stay relevant in academic circles, there will likely never be a time when it is not a requirement to continually update knowledge about known programming languages, or pick up some new ones.
As this is a never ending journey of discovery, it stands to reason that facilitating and smoothing this journey is a worthwhile effort that will continue to pay dividends long into the future.
To help pave the way for future programmers, and assist experienced veterans, this thesis seeks to explore how programmers experienced with at least two programming languages can use that knowledge to aid in their learning of a new one, by transferring the concepts they're already familiar with to a new context. Specifically, the thesis looks at intermediate programmers with experience programming in at least two programming languages with an object-oriented focus, and look at their journey learning a functional programming language.
To aid in the exploration, this project will employ the MPLT[2], a validated model for predicting transfer through the comparison of syntactic and semantic similarity of a concept in a New Programming Language (NPL) to a Previous Programming Language Previous Programming Language (PPL). The model predicts students' transfer along three different categories: A True Carryover Concept (TCC) is a concept that is both syntactically and semantically similar between NPL and PPL, a False Carryover Concept (FCC) is a concept where the syntax is similar but semantics are not, and an Abstract True Carryover Concept (ATCC) is a concept that have similar semantics, but different syntax. Each of the different transfer categories comes with its own set of predictions and significance for transfer, all of which will be explored further in later chapters.
As the MPLT has previously only been validated in the context of novice programmers transferring from Python to Java, or vice versa, which are two very similar Programming Language (PL)s, this project also aims to expand upon its use cases by validating it in the context of intermediate students transferring their knowledge from the Object-Oriented (OO) paradigm to the Functional Programming (FP) paradigm.

1.1 Hypothesis

The hypothesis of this thesis is: The MPLT transfer categories are in evidence when intermediate students from an OO background transfer to FP, and exploring this transfer with students and their teachers could improve the process.

1.2 Research questions

In order to explore the various relevant areas of transfer and the MPLT, 7 research questions have been created.

1.2.1 What outcomes would be predicted by the MPLT for an intermediate student's first encounter with FP

To test the efficacy of the MPLT in the context of transfer from OO to FP, it is first necessary to explore what predictions can be made with the MPLT from an OO PL to a FPL.

1.2.2 How does the intermediate students' experience transferring to FP match the predictions of the MPLT?

After the predictions have been made, they will need to be tested to see how well they stand up to actual conditions. The MPLT has been validated in the context of novice programmers transferring from OO Python to OO Java, so it is necessary to observe its accuracy when predicting transfer for intermediate programmers transferring from OO to FP.

1.2.3 What is the effect of students with prior programming transfer experience on the predictions, effectiveness and usefulness of the MPLT by comparison with novices transferring?

The MPLT has been previously validated in studies conducted with novice students transferring from their first PL to a second one. However, the students in this study will have transferred from a PPL to a NPL at least once, if not multiple times. To study the predictive power of the MPLT, it is important to also investigate its effect on students with prior transfer experience.

1.2.4 How does having an OO background impact students' experience transferring to FP?

FP and OO are rather distinct paradigms, with very different, often contradictory approaches. It is of great interest to observe what impact years of OO experience will have on students transferring to FP, to observe to what degree teachers will have to address the difference in paradigms.

1.2.5 What do teachers think about transfer and what is their current practice?

The primary motivation of this thesis is to aid in improving FP teaching, and a large part of this effort will consist of highlighting to lecturers any benefits of transfer-based learning and how to facilitate it. In order to make any kind of improvements to the current situation, it is first necessary to uncover the status quo, to explore the lecturers' current understanding and practice regarding transfer.

1.2.6 Are there issues of transfer to FP beyond the MPLT?

The MPLT is a model that predicts students' conceptual transfer based on syntax and semantics. However, some elements of transfer may go beyond mere syntactical and semantic differences. The aim here is to investigate this aspect. If such elements are found, they will need to be covered separately.

1.2.7 How could exploring transfer aid in teaching FP?

Part of the stated goal of this thesis is that exploring transfer will aid the process of transferring to FP. It is thus important to address whether and how this exploration will benefit FP teaching.

1.3 Study setting

This study involves teachers and students in two courses on functional programming, one at the University of Glasgow, using Haskell, and another at the University of Oslo, using Scheme.

1.4 Thesis structure

Following this introduction, the 2. chapter of this thesis details the background information relevant to know for the remainder of this thesis, in particular prior research and the different paradigms and PLs used. These include so-called guess quizzes to validate MPLT transfer, and interviews with students, teaching assistants and lecturers, all of whom are involved with FP courses.
The 3. chapter explains the methodology of the thesis: what methods have been used, why they were chosen and how they were used.
After the methodology, chapter 4 answers RQ1 by discussing what predictions can be made with the MPLT for Scheme and Haskell. These predictions are then used in chapter 5 to construct the guess quizzes for testing the validity of these predictions.
Chapter 6 introduces the key findings found during this the analysis of the guess quizzes and the interview transcripts, which are then discussed in chapter 7, before chapter 8 ties it all together in a conclusion.

Chapter 2

Background

This chapter looks at previous research into PL transfer, intermediate students and the MPLT, in addition to providing an introduction into the paradigms and PLs touched upon in this thesis.

2.1 Literature review

The first three sections of this literature review argues for the focus of this thesis on the issue of intermediate students in CS research. Specifically the issue that the majority of research into how programmers learn programming languages has focused on either novice or expert programmers, with relatively little attention paid to those in the middle ranges of experience. The fourth section details research into PL transfer, which is what happens when learners move from one PL to another, before the last section presents the MPLT, a validated model for predicting PL transfer.

2.1.1 Novices learning Functional Programming

Compared to the amount of research into intermediate student programmers, there is a large body of computer science research that looks into how novice programmers learn their first or second programming language, with multiple studies examining how novices learn, what language they should start with, what problems they're likely to face and so on. A simple query turns up thousands of results, like Joosten et al (1993)[3], Tirronen et al (2015)[4], Lahtinen et al (2005) [5], etc. These and many more studies have been conducted in an effort to help lower the barrier for entry into the world of programming. However, once the students have passed the barrier and begun learning how to program, passing into the ranks of the intermediate students, there is comparatively little research.
Another issue worth noting is that going by this wealth of research, one would think that the challenges facing new programmers would be well understood by now. This seems to not necessarily be the case. For instance, when listing the greatest challenges facing students seeking to learn FP languages, Tirronen et al (2015)[4] found that many teachers and researchers mention recursion either first or close to it. Yet, the article also found that this view might actually be incorrect. Their study looks at the common types of mistakes made by novices when learning the Functional PL Haskell as their first PL, by providing them with a coding environment that logs

compilation errors the students come across when coding various tasks. According to their findings, type-mismatch and scoping issues were among the most numerous and time consuming errors faced by students, with little evidence to show that students confused recursive base cases with stopping conditions, nor did they seem to struggle with lazy evaluation, another claim the authors came across.
While the study is hardly perfect (the authors themselves acknowledge that forming any kind of comprehensive view of students' minds using only their error ratios is impossible), it still highlights discrepancies between teachers' expectations and reality. Further discrepancies are seen in the international survey conducted by Lahtinen et al (2005)[5], where 559 students and 34 teachers were asked about perceived difficulties in learning programming, where the teachers systematically believed every single issue in the course content to be more difficult than how the students perceived the same. These discrepancies show a clear need for more research into how students learn programming, and while not strictly focusing on novice programmers the efforts of this project should help illuminate the actual challenges students face when learning FP. Clarifying struggles and challenges students face will help teachers in creating curricula to better address students' needs, and the thesis' focus on intermediate students rather than novice students will also contribute to the relatively small body of research into this level of students.

2.1.2 Expert programmers learning new Lanugages

Just as for novices, there have also been numerous studies conducted looking at how expert programmers add new programming languages to their toolbox, e.g. Shreshta et al (2020)[6], Meyerovic and Rabkin (2013) [7], etc..
Having already learnt a multitude of different languages, frameworks and patterns over the course of their careers, one might expect that picking up a new PL would be easy. Yet studies show that this is rarely the case, even when the new language is comparable to languages the programmer is already familiar with. True, the experts have rather more experience learning new PLs and tend to set better plans for their own learning process, but the evidence shows that their extensive experience actually gets in the way surprisingly often.
Shreshta et al (2020)[6] noted that developers are often held back by old habits in their preferred languages interfering with their new efforts, for instance thinking that indexes start at *a[1]* rather than *a[0]*, writing types for variables in Python and a slew of similar issues that seem small individually, but are still time-consuming and frustrating. They also noted that experts that are used to working in a certain way in their known languages are frequently thrown off when concepts they expect to work the same turn out to be completely different, or when the patterns they're used to work with do not quite fit with the new PL, which is doubly true when they're dealing with paradigm shifts.
This could be the Python-programmer having to deal with Julia's structs or the developer used to C# who struggles with Rust's unique ownership feature. The greatest identified challenge appears to be with learning new PLs that have little to no shared paradigms with what they're familiar with.
The authors[6] also noted that experts would benefit greatly from documentation that explicitly facilitates language transfer, and recommend being more intentional when looking at the syntax, semantics and pragmatics of a new language.

As we can see that even experts struggle with PL transfer, exploring this area further holds great potential for improving the learning experience for all programmers who wish to add more PLs to their tool belts.

2.1.3 Intermediate student programmers

As shown in the previous two sections, there has been plenty of research into how beginner and expert programmers learn a new PL.
However, there have been relatively few attempts to understand the difference in how intermediate students learn and improve their fledgling skills compared to their novice and expert counterparts. Further confusing this field of study, what few articles there are about intermediate students struggle to even agree on a shared definition of what an intermediate student is.
For the purpose of this project, students are considered to be at an intermediate level if they have completed at least one or more years of structured programming courses, covering two or more languages, but not yet earned a degree.
According to studies like Kopec et al (2007)[8], Shabo et al (1996)[9] and Decker & Simkins (2016)[10], such students are reasonably competent and confident in their skills, but still have some gaps in their knowledge.
Whereas novice students have a weak grasp of both syntax and semantics of the one PL they know, intermediate students should have some experience with more than one syntax, giving them some experience with transfer already. They tend to make fewer beginner mistakes, but also begin making some more complex errors. Exactly what "complex errors" entails is difficult to effectively pin down, as the kinds of mistakes can vary wildly between individuals, but the cited studies make it clear that the way intermediate students approach a new programming language is quite different from novices in that they have less experience learning languages than the expert programmers, but still have enough to occasionally facilitate or impede their transfer efforts.
As there appears to be a dire need for more research into how intermediate students continue their programming journeys, this project seeks to contribute to this area.

2.1.4 PL knowledge transfer

There have been multiple studies exploring how having knowledge of at least one PL affects transfer to a new one.
Bower & McIver (2011)[11] look at students with experience coding in C++ as they take the leap to Java. They note that their prior knowledge mostly helped facilitate language transfer as well as having positive effects for their abilities in C++, although also showing that completely new concepts or concepts perceived to be conflicting with existing knowledge could occasionally inhibit new learning.
Bower & McIver's study compared two OO-languages, Java and C++. Similarly, Tshukudu's work (which has been integral in shaping this project) studied the transfer from two closely related languages: Imperative Python to OO Java, and vice versa, over a series of articles described in the next section.
Santos, Nystrom & Hauswirth (2019)[12] did a study going the opposite way, FP to OO, and found among other things that students with a background in Racket struggled to understand classes and if-statements, their experience with structs and cond-statements getting in the way.

Tirronen (2014)[4] actually did do research on transfer to functional programming for students with OO backgrounds, however this study was laser-focused on low-level type errors, and also failed to establish exactly which PLs the participating students knew, so it is not as helpful as one could hope.
Looking at the available academic research into transfer from one PL to another, there appears to be a distinct lack of data properly covering how students transfer from an OO PL to a FPL, which is why this projects aims to remedy this glaring hole in CS research.

2.1.5 Model for Programming Language Transfer

In order to properly study the PL knowledge transfer, it is necessary to set up a framework in order to make predictions that can be tested. Tshukudu et al (2020)[1, 13, 14, 15, 16] created a Model for studying Program Language Transfer (MPLT) which largely served as the inspiration for this project.
The model, based on Jiang's model for semantic transfer in natural languages[17], was created with the idea that the way the syntax of concepts in Previous Programming Languages (PPLs) compares to the syntax of concepts in a New Programming Language (NPL) influences the speed and ease with which learners pick up the new language. According to this model, students will transfer semantic knowledge from the PPL whenever they see matching syntax in the NPL, associating a concept they know with the similar syntax. This can be either beneficial or detrimental to the student's learning experience, as described below. Using this model, it should be possible to predict the struggle students will face when beginning to learn a NPL.
Thus, teachers aware of the MPLT should be able to use the model to create curricula and study plans that address issues before students face them, avoiding unnecessary confusion and incorrect assumptions, while also allowing teachers to spend less time on concepts students are going to find trivial.
While this model has been validated in prior studies, those studies primarily focused on novice students transferring between relatively similar languages, eg. from Java to Python or vice versa. It has not been sufficiently tested with more distinct languages or paradigms, nor with more experienced students, which is why it is of interest to attempt using it to model transfer for intermediate students going from OO java to the FP languages Haskell and Scheme.
The model maps concepts from the PPLs to the NPL into three different categories:

True Carryover Concepts (TCCs)

Concepts that are syntactically and semantically similar are TCCs. These are concepts like *if*-statements in Java and Python, which have small cosmetic differences but otherwise identical overarching logic. As the concepts are so similar to one another, experience with the concept in one language lends itself to learning the concept in another.
Tshuduku's studies into the transfer between Python and Java shows that students experienced with concepts like string concatenation and *while*-loops in one of the languages would very likely score well for the conceptual equivalent in the other language.

False Carryover Concepts (FCCs)

Concepts that are similar syntactically but differ semantically are called FCCs. An example FCC between Java and Python found by Tshukudu[2, p. 71] was type coercion, where in Java it is perfectly acceptable to write *System.out.println("Hello"+3)* and Java will implicitly convert 3 from integer to string, whereas writing *print("Hello"+3)* in Python will lead to a type error.
These kinds of concepts create a minefield of potential misunderstandings for students, as they will see these concepts and erroneously believe they've understood how they work without looking deeper into their inner workings, causing a negative impact on their transfer.

Abstract True Carryover Concepts (ATCCs)

Concepts that are syntactically different but semantically similar are labeled ATCCs. At first glance these concepts appear to have no relation despite actually functioning similarly, like Tshukudu's example[2, p. 71] of the data structure concept. In Python, all it takes to create a data structure with data fields assigned with values is f.ex *p1 = {'age':3, 'name': 'Agnes'}*. However, in order to create the equivalent structure in Java we would need to write significantly more code: .

```
public class Person
{
    int age;
    String name;
    public Person (int ages, String names){
        age=ages;
        name=names;
        }
    public static void main(String[] args){
        Person p1 = new Person(3, "Agnes");
    }
}
```

If no particular action is taken to make students aware of this similarity, prior research shows that there does not naturally occur any conceptual transfer with an ATCC. This happens because students mistakenly believe they've encountered a brand new concept and need to have the similarity explicitly pointed out to them in order to bridge the gap between the new concept and the previously known one. Once the connection has been made however, the transfer should work similarly to TCCs as the students do know how the concept works, they simply have to get used to implementing it in a slightly different way.

2.1.6 Summary of literature review

This review has focused on the three main areas of research relevant to this project: Intermediate students, PL transfer and the MPLT.
The review began by looking at studies into programmers at different levels of experience, noting how experts and novices have gotten far more attention in academia compared to intermediate students. Intermediate students are found to have a stronger grasp of syntax than novices, in multiple languages, but less programming

experience than experts.
The review moved on to look at research surrounding PL transfer, with several studies into novice students' transfer between languages of various paradigms, such as OO to Imperative programming, FP to OO, or even just one OO PL to another. None of these studies attempted to understand intermediate students, and while there was a study into transfer between OO to FP, this study was too laser-focused on typing and too vague on students' programming experience to suit the purposes of this project.
Lastly this review discussed the MPLT, a validated model for predicting PL transfer which will be used in this project. The MPLT divides concepts in a NPL into three different categories: TCCs, FCCs and ATCCs depending on syntactic and semantic familiarity. While this model has been validated, it has not been sufficiently tested with intermediate students' PL transfer from an OO language to an FP language, which is why this project is of great interest to the scientific community.

2.2 Programming languages

This thesis deals with four PLs: the object-oriented Python and Java, which students will have learnt as part of their introduction to programming, and functional Scheme and Haskell, which the students will learn through the course of the studies this thesis presents.

2.2.1 Object-Oriented Programming Languages

Object-Oriented programming languages are derived from the family of imperative languages, which fundamentally consist of instructions that act on the program's state. OO simulates and interacts with data by creating virtual objects to interact with, designing classes with characteristics that can be changed to update the program's state.
Based on the blueprint provided by a class, the program can create objects with methods to interact with itself and surrounding objects.
The key principles of OO are

- **Encapsulation**: The implemention of an object and data not necessary for the user is hidden from view.
- **Inheritance**: Once a class has been designed with characteristics and methods, a new class can be created that inherits these characteristics and methods, reusing the functionality of the parent class.
- **Polymorphism**: A object that inherits data from a parent class can be treated as an instance of its own class or that of its parent, allowing the object to pass interfaces that accepts either one.

Main benefits of the OO paradigm is its reusability, flexibility and scale-ability. Once a class has been created, any number of objects can be used with that class or its sub-classes as a template.

Python

The Python programming language was released in February 1991 by Guido van Rossum, and intended as a successor to the PL ABC.[18].
Python is a very popular, multi-paradigm PL supporting imperative, OO and FP styles.[19]
The PL is dynamically typed, using type-inference, and it governs programming blocks through indentations. Its core philosophy is to be beautiful, simple and explicit, favoring complex over complicated code, with a focus on readability.
Python is widely considered to be a very beginner-friendly PL, and is used by both the University of Oslo and the University of Glasgow to provide new students with an introduction to programming.
While Python is multi-paradigm, supporting FP, both universities focus on its imperative and OO qualities. The introductory course teaching Python is called IN1000: Introduction to Object-Oriented Programming.
Due to this focus on OO in teaching Python to the students investigated in this project's studies, this thesis considers Python an OO PL in this context.

Java

The Java programming language is an OO PL that was developed by James Gosling for Sun Microsystems (later aquired by Oracle) and released in 1995.
Java is a general-purpose PL designed to have as few dependencies as possible. The PL is intended to let application developers *write once, run anywhere (WORA)*[20]
The syntax of Java was influenced to a large degree by C and C++. It was built as an OO PL, requiring all data to be contained within classes, treating every data item as an object (excepting primitive types like integers, booleans and chars).
Java is a statically typed language, requiring all variables to have a defined type upon declaration.

2.2.2 Functional Programming Languages

The functional programming paradigm is an approach to software engineering where computer programs are expressed through pure functions, functions that do not interact with mutable state.
Key principles of FP are:

- **Pure functions**: Functions do not interfere with global state, and can be said to have no side effects.
- **Higher-order functions**:In FP, functions are treated as first-class citizens. They can be used as arguments or return values for other functions.
- **Immutability**: Contrary to OO, FP avoids the use of mutable data and shared states, with a greater focus placed on the result of a function rather than the processes that make it up.
- **Recursion**: As loops necessitate mutable state to update for each repeat, which breaks with the functional paradigm, FP programs rely extensively on recursion instead, using functions that call themselves to calculate the results of expressions.

Main benefits of using the FP paradigm are the modularity and simple concurrency afforded by the lack of shared, mutable state. The result of a pure function relies solely on its parameters, independent of other processes that might run in parallel.[21]

Scheme

Scheme was created in 1975 by Guy Steele and Gary Sussman, and presented in a series of articles now known as "the Lambda papers"[22].
Their goal was to clarify non-recursive control-structures in a recursive PL like Lisp, explain how to use these structures independent of issues like pattern matching and data manipulation, and to have a simple concrete experimental domain for certain issues of semantics and style.
As a dialect of Lisp, Scheme has an interesting syntax that at first glance doesn't bear much resemblance to the majority of computer languages, among other things using prefix-notation in addition to the famously all-pervasive parentheses, yet there are similarities in the concepts that allow us to take advantage of PL transfer both when teaching and learning the language.

Haskell

As detailed in *The History of Haskell: Being lazy with class*[23], in the late 80s ideas such as lazy evaluation had started to gain a lot of traction, and there were a plethora of functional languages like Miranda, Orwell, LML and Ponder, that were created to take advantage of it. In fact, there arose so many new FPLs during this time that researchers in the field of functional programming started to complain that they were hampered by the sheer variety.
To help remedy this problem, a special committee was set up in 1987 with the goal of designing a common FP language that could consolidate the best of the rest. The result was Haskell, the first version of which was released 1. April, 1990.
Haskell is a lazily-evaluated, purely functional PL with a strong type-system. The Haskell syntax bears strong resembles to SML and Miranda, and like Python it interprets significance through block placement and white space. Haskell also infers types on variables, although its strict type-system allows for far less flexibility in changing the type of a variable once defined than Python's dynamic typing.

Chapter 3

Methodology

This chapter concerns itself with detailing the methods used in data gathering for this thesis study.
Included are explanations of the methods themselves, as well as why they were chosen.

3.1 Guess quiz

To answer RQ2: *How does the intermediate students' experience transferring to FP match the predictions of the MPLT* requires gathering quantitative data on students' ability to transfer their PL knowledge from PPL to NPL. To accomplish this goal, the FP students were asked to take an anonymous guess quiz based on the predictions made with the MPLT that were discussed in chapter 4. A guess quiz is a quiz where students are presented with code-examples from a PL they're not yet familiar with, and then asked to guess how the code works, and what they believe the outcome will be.
This approach was used in the studies[1] that validated the MPLT as a model for predicting transfer, so it was decided to use it for this study as well, to ensure the replicability of the study.
By giving a guess quiz, the idea is to observe how students make use of their prior knowledge to comprehend new concepts. Their educated guesses help glean important insights into their instinctual transfer, and help verify predictions about what concepts the students will find easy, what they will misunderstand and what they're going to find alien.
Answers for all guess quizzes were expected to be given in free-text format rather than f.ex multiple-choice. The main reason for this was for the flexibility in answers this format allows, enabling observation of not only what concepts students transfer, but also how they transferred them and what enabled them to make the connections they did. However, a weakness with this approach was that it required more work in analyzing the quizzes, as detailed in a later section.

3.1.1 Glasgow guess quiz

At Glasgow university, students taking 'Functional Programming (H) COMPSCI4021' were asked to participate in a guess quiz at the start of the semester, drawing answers from 28 students. This guess quiz consisted of 5 questions asking students to guess the function of various pieces of a Haskell code example.

3.1.2 UIO guess quiz

The students taking 'IN2040: Functional Programming' at UIO were given one guess quiz in their first lecture, with 65 students answering. The quiz was administered before any PL teaching had begun, and most had thus not seen any Scheme before. The students were asked some simple, non-identifying questions about their prior programming experience, including whether they had experience prior to starting at UIO, and what PLs they were most comfortable with. The students were asked to answer questions about 4 different Scheme code examples.

3.1.3 UIO novice programmers guess quiz

In addition to hosting a guess quiz for the intermediate students taking the UIO FP course, a small number of novice students were also asked to participate.
These students completed the same guess quiz as the intermediate students, in a separate session later in a year. At that point, these students had half a year of experience programming in Python.
The reason for inviting novice students to participate was to compare and contrast their results with those of the intermediate students, to better observe how intermediate student transfer differed from that of novice student, for the purpose of answering RQ3: *What is the effect of students with prior programming transfer experience on the predictions, effectiveness and usefulness of the MPLT by comparison with novices transferring?.*

3.1.4 Analysis

In order to analyze the results of the free-form answers to the quizzes, each question was first broken down into its constituent concepts. Then the answers were corrected based on the students' mention and correct use of that concept.
While the results are primarily treated as quantitative data, the qualitative efforts involved should not be disregarded.
As the free-form answers allowed students to phrase their explanations however they see fit, responses were rarely as cut-and-dry and uniform as one could hope.
The variety in possible answers required some interpretation during analysis. For example, an answer that does not explicitly mention the *-operator in *(* 5 5)*, but mentions that the result would be 25 would still be marked as correct as the student has shown an understanding of the concept by correctly guessing its outcome.
Understanding of a given concept was put into three categories: 'Correct', 'Incorrect' or 'Didn't answer'.
Answers were marked as 'Correct' if the concept was correctly explained, or if their explanation of how the code would run required a correct understanding of the concept. Answers were marked as 'Incorrect' if the concept was misidentified, or if the their explanation of how the code would run required an incorrect understanding of the concept. Answers were marked as 'Didn't answer' if the concept wasn't mentioned at all, or if their explanation didn't sufficiently show whether or not they actually understood the concept.

3.2 Guess quiz difficulties

No method is without its flaws, and the guess quizzes are no exception. The method itself has its limits, and gathering results aren't without always unproblematic either. What follows is a short summary of issues with using guess quizzes.

3.2.1 Hard-to-test concepts

While it is in many instances relatively straight-forward to measure whether a student understands a concept using a guess quiz, some concepts are trickier to measure. This could be because the semantics of the concept relies on other concepts the students might not understand, or because the context of the quiz requires the concept to be shown in a misleading way.

Scheme-example: List

An example from Scheme is the list-concept, where understanding that there is a difference in semantics require students to understand more concepts in order to grasp.
The concept does not share syntax with either Python or Java, but is implemented as a linked list, which students at UIO are expected to know as they are tasked with implementing the data structure in Java in a previous course. Scheme's lists were thus predicted as an ATCC. However, to understand this implementation, the students also have to understand *car* and *cdr*, Scheme's expressions for extracting the head of the list and the head's next node respectively. These concepts are both predicted as ATCCs, as they have novel syntax but students' have experience with their semantics.
The syntax of a list in Scheme looks like this:

(list 1 2 3 4 5)

Which leaves us in the situation where the students are very likely to understand that the concept *is* a list, and which they will likely relate to Python's list or Java's array purely due to the obvious keyword, despite the syntax being different from what they are used to. However, they are not likely to expect that the list is implemented differently from what they are used to, and thus will not comment on its implementation sufficiently to be marked wrong.
As they are not likely to comment on the concept beyond that it is a list, the results of the guess quiz will likely see the majority of students marked as answering correct for this concept, despite its prediction as an ATCC.

Haskell-example: If-expression

Haskell's if-expression is an example of a concept where students are likely to be marked as correct due to the context of the code, even though they likely misunderstand the concept's semantics.
Similar to Scheme's *if*-expression, Haskell's if-expression is implemented as a *ternary*-operator rather than like Java's and Python's *if*-statements. The expression looks syntactically very similar to Python and Java:

if <predicate> then <consequent> else <alternative>

The semantic difference is that, unlike in Java and Python, the *else*-block is not optional. Since the syntax is similar but the semantics are different, this concept is predicted as an FCC.
However, it is very difficult to understand from a correct Haskell code-example that a block that is present *has* to be present. It is thus unlikely that students are going to realize the difference in semantics from the guess quiz example. This means that they are likely to simply mention that there is an *if*-expression present in the code, but not likely to comment sufficiently on the implementation to be marked as wrong. Similar to Scheme's list-concept, the students are thus likely to be marked as correct for this concept, despite its prediction as an FCC.

3.2.2 The problem with non-answers

Over the course of the guess quiz, students are faced with a multitude of different programming concepts, and occasionally they fail to mention a few. This is expected of course, but leads to a tricky dilemma: analyzing the meaning of a failure to answer. A student not mentioning a concept could be interpreted in any which way. Perhaps they didn't see the concept. Perhaps it was completely alien and such they had nothing to write about it. Or perhaps they found the concept so glaringly obvious they saw no need to explain its meaning.

3.3 Student interviews

To get a more thorough understanding of the reality students face when transferring to FP, a handful of students were asked to participate in interviews about their experience. This approach was chosen in order to gain deeper insight into how the students learn FP for the first time, how they transfer their knowledge from PPLs as well as to understand how aware they are of their own transfer.
Interviews are a flexible, efficient way of achieving those goals when done correctly, as it can gather large amounts of information on the subject matter at hand, as well as allowing the interviewer to ask follow-up questions on the spot that might help illuminate a given topic further than a static questionnaire could ever hope to.

3.3.1 Participants

The study interviewed 5 student participants chosen among students taking the course IN2040: Functional Programming at the University of Oslo in the fall of 2022, teaching Scheme.
To qualify for the course, students had to have had at least one year of programming courses, and would have learned Python and Java among other PLs.
The interviews were all conducted during October 2022, two months into the course.

3.3.2 Interview structure

The interviews were semi-structured, each lasting approximately 30 minutes wherein participants were asked a range of questions about their backgrounds, their learning experience, their thoughts on FP, as well as whether they could relate the concepts they'd seen and learned to concepts they knew in previously learned PLs.

The students were also asked to complete three simple code-comprehension tasks created for these interviews, as well as solve a simple programming challenge were they were tasked with finding the length of a list.
The reason for asking the students to solve tasks was to get a glimpse of their problem-solving approach first-hand. This allowed for a better perspective of how they planned and reasoned when solving tasks, how aware they were of the different concepts that cropped up, and to see how close their explanations of how they saw FP aligned with how they actually solved FP tasks.
All 5 interviews were conducted in Norwegian, then translated to English during the transcribing process.

3.4 Interviews with Teacher assistants

The Department of Informatics at the University of Oslo employs a system where students who have passed a course previously can be hired to assist in teaching the course at a later semester.
This is a part-time job offered only to current students, typically students who achieved a high grade in the course themselves. The student is expected to prioritize their studies so that the job does not detriment their education. Under this system, the lecturer is responsible for the curriculum and theoretical aspects of the course, whereas the TAs typically lead one or two weekly sessions for a smaller group of students taking the course, using their prior experience to help the students navigate the curriculum, answer practical questions and correct assignments.

3.4.1 Reason for interviewing TAs

Due to the practical nature of the TA position, TAs typically have a more hands-on perspective of the students taking the course. They observe the students while they're learning, answer their questions and are often asked to help troubleshoot faulty code and clarify misconceptions.
This places the TAs in a position to truly see the students' progression and observe whether any transfer is going on, giving them a closer look at students' day-to-day than a lecturer could hope for.
It is of great importance to understand how conscious TAs are of transfer; if they've observed it in their students or if they take advantage of it when they teach.

3.4.2 Interview structure

Similar to the student interviews, the TA interviews were semi-structured and lasted approximately 30 minutes each. The participants were asked about their backgrounds, their awareness of transfer, whether they'd observed students transferring PL knowledge, as well as whether and how they employed transfer in their teaching.
The TAs were also asked to complete the same coding tasks as the students, to see how well they understood the course material and whether their thinking, as more experienced FP programmers, differed from that of their students.
All three interviews with TAs were conducted in Norwegian, then translated to English during the transcribing process.

3.5 Interviews with lecturers

This project has focused on two FP courses: 'IN2040: Functional Programming' at the University of Oslo which teaches Scheme, and 'Functional Programming (H) COMPSCI4021' at the University of Glasgow.
The course at UIO was recently taken over by a new lecturer who's first semester teaching it was the same semester this study was conducted on the course, and as such is still coming to terms with its contents. In addition to this lecturer, the study also interviewed a lecturer at UIO who's been heavily involved in IN2040: Functional Programming in the past as a TA.
Lastly, the study interviewed the two lecturers responsible for the course at Glasgow, who have been teaching the Haskell course there for many years. They've also created a Massive Open Online Course (MOOC) 'Functional Programming in Haskell' which has garnered popularity.

3.5.1 Reason for interviewing lecturers

This section aims to answer RQ5: *What do teachers think about transfer and what is their current practice?*. As this study seeks to uncover how transfer impacts students with OO backgrounds learning FP, in the hopes that the results can help improve the teaching of FP courses. It would be a strange oversight to try to present the findings of this study without also seeking context from the lecturers teaching them. The lecturers have many years of experience with programming, both FP and otherwise, and are a wellspring of information about the paradigm, its uses, how it's been taught over the years as well as how the students have responded to that teaching. In order to help them improve their approach to teaching for transfer in the future, it's important to understand how they approach transfer at present.

3.5.2 Interview structure

The interviews with the lecturers were held in a semi-structured format, but unlike the students and TAs, the lecturers were not asked to undertake code tasks in order to focus more on the theoretical aspects of their work. The interviews lasted between 45 minutes to an hour. The lecturers in Oslo were interviewed in person, whereas the lecturers in Glasgow were interviewed over Zoom. The interviews with all three lecturers currently teaching FP were conducted in English, whereas the interview with the lecturer who had previously been involved with the course was conducted in Norwegian and translated to English during the transcribing process.

3.6 Thematic analysis

The interviews were recorded, with participants' consent, and later transcribed. In order to analyze the interviews, thematic analysis[24] was the chosen approach, based on the technique's flexibility . This is a technique that allows for analyzing large amounts of qualitative data by looking at and codifying patterns that emerge from said data.
First, a small number of questions were designed as a source of inspiration for the analysis. Example questions:

- How has student's Object-Oriented background impacted their transfer to Functional programming?
- Do teacher's assistants take advantage of transfer?
- Do lecturers use transfer when creating curriculum and teaching?

Then, with these questions as inspiration, the various statements in the interviews were codified. Themes that emerged during the codifying process were noted down for later use, and included:

- Transfer awareness
- Teaching for transfer
- Transfer impact

Next, relevant quotes found were pulled out into a separate document, and then sorted under various labels, like:

- Students' awareness of own transfer
- Transfer depends on prior knowledge
- Instructor's role in transfer

Once all quotes were labeled, the different labels were then sorted under the various themes that were found during the codifying process.

Chapter 4

Predicting transfer

This chapter intends to answer RQ1: *What outcomes would be predicted by the MPLT for an intermediate student's first encounter with FP.* Answering this question requires looking into the various concepts in Scheme and Haskell that transfer, and categorizing them based on their syntactical and semantic similarity to concepts in the students' PPLs.
This chapter will illustrate the process by taking a look at one example concept from each category for both NPLs: Scheme and Haskell.

4.1 How transfer is predicted

The MPLT sorts its transfer predictions into three categories, based on the concept's syntax and semantics.

4.1.1 Transfer categories

Concepts that have similar syntax and semantics are predicted as TCCs. Students are expected to see these concepts as familiar to what they already know, and because they work the same way as they are used to the impact on their transfer will be positive.
Concepts that have similar syntax but different semantics are predicted as FCCs. Like the TCCs, students are expected to see FCCs as familiar to what they already know, but because they actually work differently these concepts will have a negative impact on students transfer.
ATCCs are concepts that have dissimilar syntax, but similar semantics to a concept the students are already familiar with. Due to the unfamiliar syntax, the students will perceive these concepts as novel, and will not transfer their conceptual knowledge to these concepts.

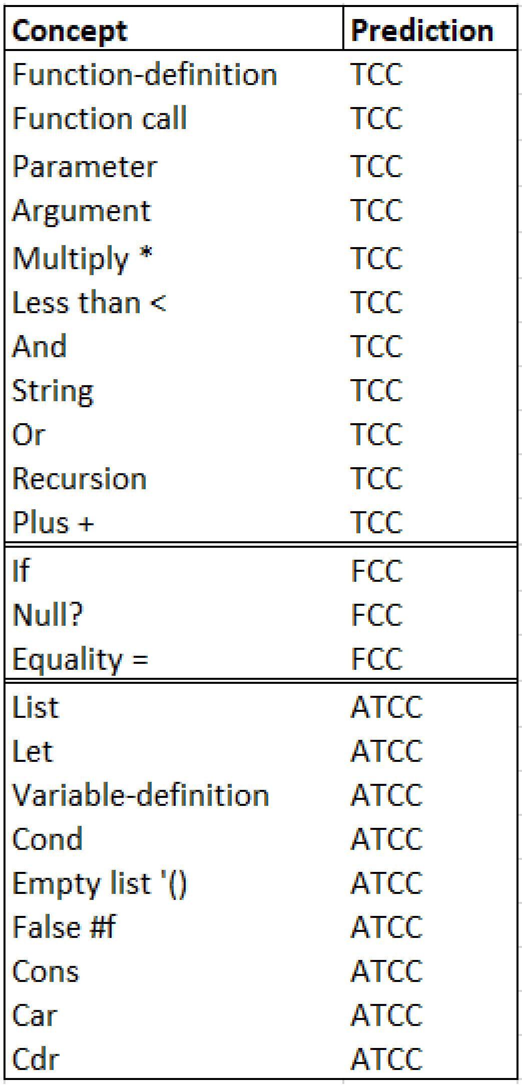
4.1.2 Predicting transfer

In order to predict conceptual transfer to a NPL, the first step is to pick a concept to predict.
The concept can be anything the NPL has that can be related to a concept in the

PPL, be it an abstract concept like recursion, or a small building block like an integer.
Once the concept has been chosen, look at how it relates to a similar concept, or concepts, in the PPL. Does it look the same? Does it work the same way?
When looking at syntax, keep in mind that the concepts do not have to be syntactically identical, just similar enough that students will be able to make a connection. Using the same keyword or operator has shown to be enough to elicit transfer in previous studies[16].
A possible challenge for researchers when predicting transfer based on semantics is that students often do not have the same deep understanding of how a concept is implemented. When predicting students' semantic transfer it is recommended to attempt looking at the concept from the perspective of the students and consider how they perceive a concept to work.
As long as the concepts work the same way on a surface level, students will likely perceive them to be functionally the same.

4.2 Predicting transfer from Java/Python to Scheme

Scheme is part of the Lisp-family of PLs, and as such employs prefix-notation and a plethora of parentheses, leaving the PL quite distinct from the common PLs students will be familiar with.
Due to these distinctions, there's no 100% syntactical matches between Scheme compared to Java or Python. However, assuming students can overcome these differences, similarities occur that the MPLT can help predict transfer of. Some of these similar concepts are exemplified in the below table.

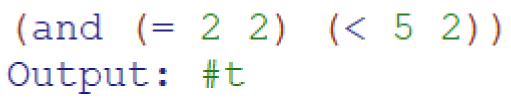


Concept	Prediction
Function-definition	TCC
Function call	TCC
Parameter	TCC
Argument	TCC
Multiply *	TCC
Less than <	TCC
And	TCC
String	TCC
Or	TCC
Recursion	TCC
Plus +	TCC
If	FCC
Null?	FCC
Equality =	FCC
List	ATCC
Let	ATCC
Variable-definition	ATCC
Cond	ATCC
Empty list '()	ATCC
False #f	ATCC
Cons	ATCC
Car	ATCC
Cdr	ATCC

Figure 4.1: Scheme predictions

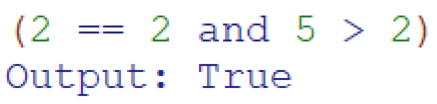
One concept from each of MPLT's categories - TCC, FCC and ATCC - have been chosen to illustrate how the different categories of the MPLT are predicted.

4.2.1 TCC - Logical operators: *and* and *or*



```
(and (= 2 2) (< 5 2))
Output: #t
```

(a) Scheme and-example.



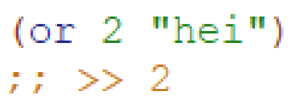
```
(2 == 2 and 5 > 2)
Output: True
```

(b) Python and-example.

Figure 4.2: TCC: *And*-expressions

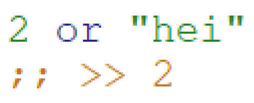
Apart from Scheme's prefix-notation and the overabundance of parentheses, *and* and *or* both looks and works the same way between Python and Scheme.
In both PLs, the constructs work to combine the truth-values of multiple expressions into one, where *and* returns true if every expression evaluates to true, and returns false ($\#f$) as long as at least one expression evaluates to false, whereas *or* returns

true if at least one expression evaluates to true, and false if all of the expressions evaluate to false.
The main difference between the expressions is that due to Scheme's prefix-notation Scheme's *and*- and *or*-expressions can combine an arbitrary number of arguments, even zero or one, whereas Python's are fixed at two arguments per operator. Their



```
(or 2 "hei")
;; >> 2
```

(a) Scheme or-example.



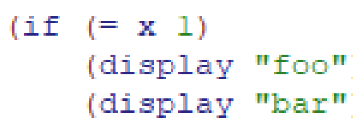
```
2 or "hei"
;; >> 2
```

(b) Python or-example.

Figure 4.3: TCC: *Or*-expressions

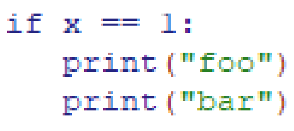
similarity even goes beyond the obvious, as not everyone are aware that Python accepts more than True and False as boolean values.
The same goes for Scheme, where every value other than false (*#f*) is accepted as a true value.
This means that the expressions will accept other types, such as integers, floats and strings, as true values, and even return these same values. *and* will return the last true value as long as all arguments are true, and *or* will return the first true value it sees. Should there be a false argument for the *and*-operator, or no true-values for the *or*-operator, both will also return their PL's equivalent of false instead.

4.2.2 FCC - If-expression



```
(if (= x 1)
    (display "foo"
    (display "bar"
```

(a) Scheme if-expression.



```
if x == 1:
    print("foo")
    print("bar")
```

(b) Python if-statement

Figure 4.4: FCC: If-statements

The if-expression in Scheme is a rather clear example of an FCC between Java/Python and Scheme, being syntactically almost identical to if-statements in Java, Python and many other languages, as seen in fig 4.4.
However, whereas the if-statement in Python will execute every statement in the body should the predicate prove true, printing both “foo” and “bar”, the if-expression in Scheme will print only “foo” if the predicate proves true, and “bar” should it prove false.
Despite the syntactic similarity to Java and Python's if-statements, Scheme's if-expression is semantically much closer to the ternary-operator present in both other PLs.

```
int x = 1 < 2 ? 1 : 2;
```

(a) Java ternary-operator

```
x = 1 if 1 < 2 else 2
```

(b) Python ternary-operator

Figure 4.5: Ternary-operator

The ternary-operator is a common construct found in many PLs that evaluates a predicate and returns one of two results based on whether the predicate is true or not. Scheme's if-expression will evaluate similarly to the Java and Python expressions in 4.5, where x will be initialized as either 1 or 2, depending on whether *1 < 2* evaluates to true.

4.2.3 ATCC - Cond-expression

```
(cond ((= x 2) (display "X equals 2"))
      ((> x 2) (display "X is greater than 2"))
      (else (display "X is less than 2")))
```

(a) Scheme cond-example.

```
(x == 2){
   System.out.println("X equals 2");}
else if(x > 2){
   System.out.println("X is greater than 2");}
else{
   System.out.println("X is less than 2");}
```

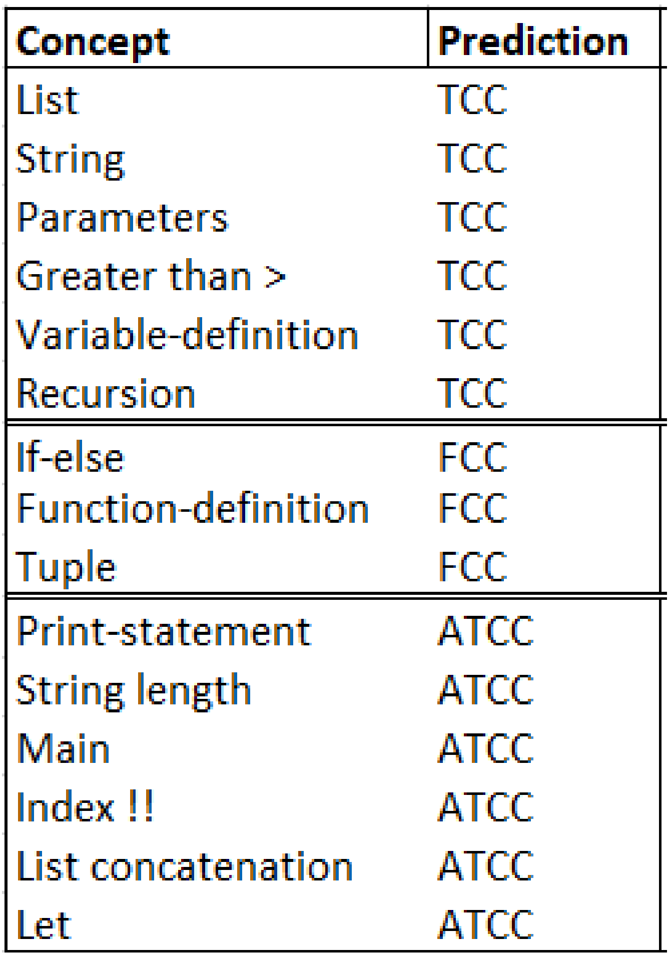
(b) Java if-else if-else-example.

Figure 4.6: ATCC: *cond* and *if-else if-else*

While the semantics of Scheme's *if*-expression doesn't work the way students are used to, Scheme *does* have a construct which works as a substitute. The *cond*-expression doesn't have an obvious syntactic similarity to Java and Python's if-statements, with a different keyword and a different way of lining up the predicates and results, the semantics are the same.
Cond takes an arbitrary number of predicates, which it will evaluate sequentially. The first predicate that evaluates to true returns its value, with an optional 'else' catch-all clause.

4.3 Predicting transfer to Haskell

Compared to Scheme, Haskell's syntax is a lot closer to what students are used to. This makes the TCCs and FCCs somewhat simpler to point out, as it is not necessary to account for oddities such as prefix-notation.
Some of the predictions found for the students learning Haskell are exemplified in the below table.



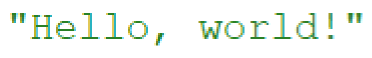
Concept	Prediction
List	TCC
String	TCC
Parameters	TCC
Greater than >	TCC
Variable-definition	TCC
Recursion	TCC
If-else	FCC
Function-definition	FCC
Tuple	FCC
Print-statement	ATCC
String length	ATCC
Main	ATCC
Index !!	ATCC
List concatenation	ATCC
Let	ATCC

Figure 4.7: Haskell predictions

Following the pattern from the section on predictions for Scheme, one concept has been picked to exemplify each MPLT category.

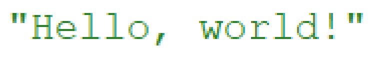
4.3.1 TCC: String

When discussing concepts that transfer, it is important to keep in mind not only big, "important" concepts like functions, classes and conditional constructs, but also the small building blocks of code, like integers, floats or chars.
Take for instance the humble string, where a sequence of characters come together to form a message.
It is so simple, yet often so important. One of the most common ways of getting outputs from a computer program is through strings, allowing the programmer to interact with their program.
Strings are so integral and beginner friendly that one of the first tasks typically given to a programmer in a NPL is to write a program that can output the string "Hello, world!" to the terminal.
Luckily, transferring strings between PLs is very simple as nearly every PL has the same implementation, as demonstrated below in 4.8.



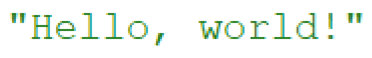
```
"Hello, world!"
```

(a) Haskell String



```
"Hello, world!"
```

(b) Python String



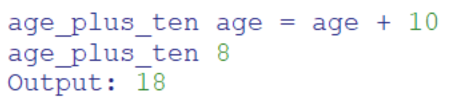
```
"Hello, world!"
```

(c) Java String

Figure 4.8: TCC: String

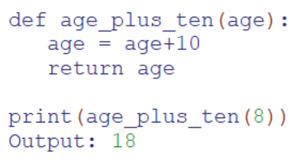
In Python, Java and Haskell (and Scheme for that matter), the syntax of a string is denoted in an identical way: a sequence of characters delimited by quotation marks. The semantics of a string does not undergo any changes either. In each PL, a string is implemented as a sequence of characters *char*.
The specific implementation might end up using either an array or a list to hold the characters, but as far as the students are concerned they work the same way.
As both the syntax and semantics are the same for both Haskell and the students' PPLs, the string concept can be comfortably predicted as a TCC for the Glasgow students transferring to Haskell.

4.3.2 FCC: Function-definition



```
age_plus_ten age = age + 10
age_plus_ten 8
Output: 18
```

(a) Haskell function-definition.



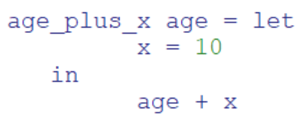
```
def age_plus_ten(age):
    age = age+10
    return age

print(age_plus_ten(8))
Output: 18
```

(b) Python function-definition

Figure 4.9: FCC: Function-definitions

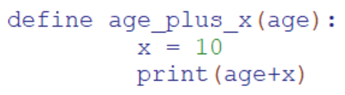
An issue for OO-experienced programmers making the transition to FP is that FPLs typically don't make much of a distinction between functions and variables, allowing functions to act as first-class citizens that can take the place of variables as the input and output values of other functions.
In Haskell, this shows itself in how the syntax of function definitions is near identical to Haskell's variable definitions, whereas Python and Java makes this distinction much clearer.
Python reserves the *define*-keyword for defining functions, whereas a function in Haskell can look completely identical to a variable, with an arbitrary number of parameters that aren't separated in any noticeable way.

4.3.3 ATCC: Let-expression and local variables



```
age_plus_x age = let
         x = 10
   in
         age + x
```

(a) Haskell let-example.



```
define age_plus_x(age):
         x = 10
         print(age+x)
```

(b) Python let-example

Figure 4.10: ATCC: Local variables

In Python and Java, defining variables locally to a class or function isn't any different to defining a variable in the global environment. Type your variables, and those variables will only live within their defined scope
Haskell doesn't make this as simple however, requiring a special construct in order to define local variables.
Students experienced with Python and Java will be used to defining local variables, and that semantic transfers to Haskell, but the additional syntactic constructs required to make use of local variables will be alien to the students.

Chapter 5

Constructing the Guess Quizzes

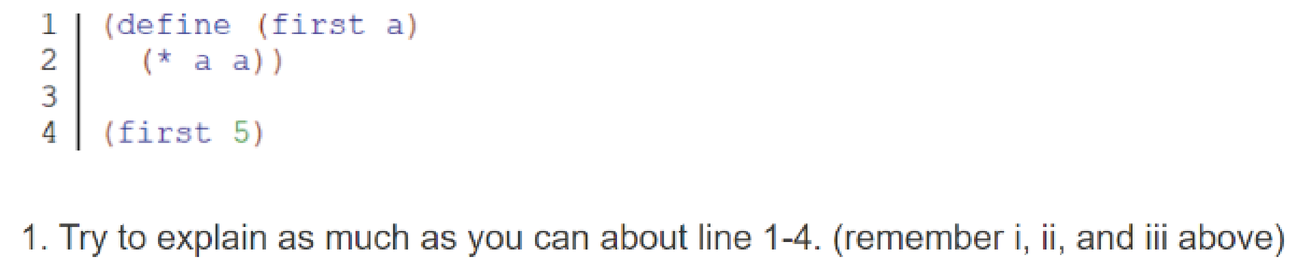
To create the guess-quizzes, the NPLs, Scheme and Haskell, were compared to the students' shared PPLs, Java and Python, to predict what kind of concepts the students were likely to transfer, and whether they would accurately identify the concepts or not.
After making these predictions, code-examples were written that implemented these concepts, allowing the students to see the concepts in context, after which the students were asked to give their assumptions about the code in free-text format.
The finished version of both guess quizzes can be found in the appendix.

5.1 UIO Guess Quiz

The students were presented with 4 questions, where each question presented them with a piece of Scheme code. The answer-format was free-form, and the students were asked to present their answers after the following format:

> *For each of the questions 1 to 4, which ask you to explain some line(s) of Scheme code, please consider your answer in three ways:*
> ***i.*** *What the line(s) of code will do when executed - what will be the outcome/result?*
> ***ii.*** *What you believe the constituent parts of the line(s) are - what computing names would you give them? e.g. function, name, parameter, argument, value - and how do they work/operate?*
> ***iii.*** *How you came to this explanation - did the line(s) remind you of code in other languages? If so, say which languages?*

The code-snippets were each between 1-6 lines, where larger questions had separate sub-questions asking about 2 or 3 lines within the larger whole.
Each code example contained a range of programming concepts, allowing students to observe them in a context that would hopefully aid their transfer in a natural fashion.

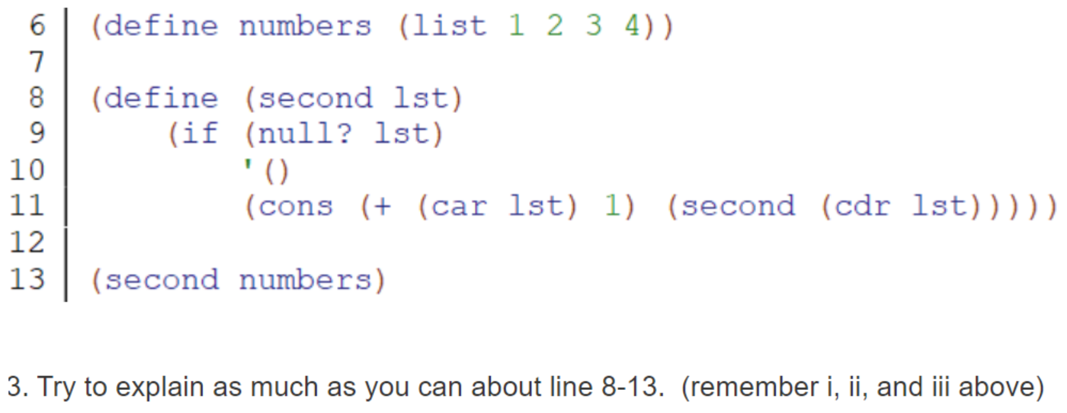


```
 1 (define (first a)
 2   (* a a))
 3
 4 (first 5)
```

1. Try to explain as much as you can about line 1-4. (remember i, ii, and iii above)

Figure 5.1: First question from the UIO guess quiz

The example in figure 5.1 was chosen as a "warm-up" question, as it was predicted to be fairly simple for students to understand.
Within the example, the code defines a function *first* which takes a parameter *a* and returns its square. The function is then called with 5 as argument.
The example presents students with the predicted TCCs function definition, function call, parameter, argument and basic arithmetic, in a way that also showcases Scheme's prefix-notation and use of parentheses. No FCCs or ATCCs were presented in this first task, to get the students a bit familiar with Scheme before showing them concepts expected to cause confusion.
Following the "warm-up", the students were shown a series of questions of greater complexity, like the example shown in figure 5.2, showing FCCs like Scheme's *if*-expression, and ATCCs like the empty list '() and the list-functions *cons*, *car* and *cdr*.



```
 6 (define numbers (list 1 2 3 4))
 7
 8 (define (second lst)
 9     (if (null? lst)
10         '()
11         (cons (+ (car lst) 1) (second (cdr lst)))))
12
13 (second numbers)
```

3. Try to explain as much as you can about line 8-13. (remember i, ii, and iii above)

Figure 5.2: Third question from the UIO guess quiz

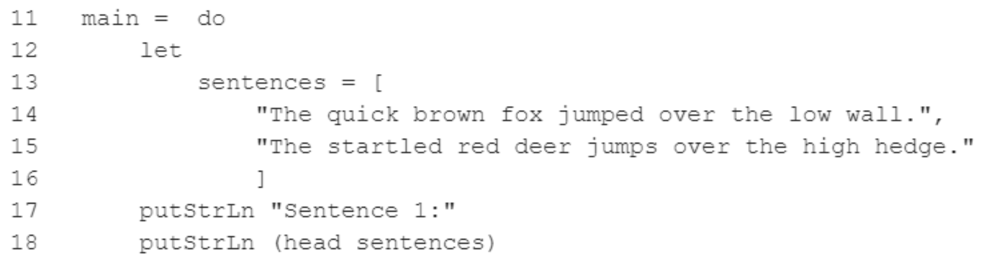
The reason line 6, *(define numbers (list 1 2 3 4))* was not asked about was because this line was covered in a previous question.

5.2 Glasgow guess quiz

The guess quiz in Glasgow was designed by the course lecturers. This guess quiz showed the students a larger piece of Haskell code, then asked questions pertaining to specific lines of that whole. Students were expected to answer the questions in free form, and asked to follow a similar format as the UIO guess quiz:

> *For each of the questions A to E, which ask you to explain some line(s) of Haskell code, please consider your answer in three ways:*
> ***i.*** *What the line(s) of code will do when executed - what will be the outcome?*
> ***ii.*** *What you believe the constituent parts of the line(s) are - what computing names would you give them? e.g. function, name, parameter, argument, value - and how do they work/operate?*
> ***iii.*** *How you came to this explanation - did the line(s) remind you of code in other languages? If so, say which languages?*

The code was 24 lines long, and contained a plethora of different concepts for students to observe, ranging from variable and function definitions, let-expressions, lists and more.



```
11   main =  do
12       let
13           sentences = [
14               "The quick brown fox jumped over the low wall.",
15               "The startled red deer jumps over the high hedge."
16               ]
17       putStrLn "Sentence 1:"
18       putStrLn (head sentences)
```

(a) *Explain as much as you can about lines 13-16. (remember i, ii, and iii above)*

Figure 5.3: Example question from the Glasgow guess quiz

This quiz-format allowed students to see Haskell-code in context, aiding their comprehension by allowing them to use code that they weren't asked about to help understand the code they were.
The code-question showed in 5.3 displayed the creation of a variable *sentences* and its binding to a list of strings, with all three of those concepts predicted as TCCs.

Chapter 6

Results

This chapter looks at the results garnered through the studies conducted during this thesis project.
The results are from the 3 quizzes: One guess quiz held in Glasgow asking questions related to Haskell, and one Scheme guess quiz held in UIO given to intermediate students, and also administered at a later date to novice students.
In addition to the 3 quizzes, there are also results from the analysis of 12 interviews: 5 held with students, 3 with teacher's assistants and 4 held with lecturers.

6.1 Guess quiz

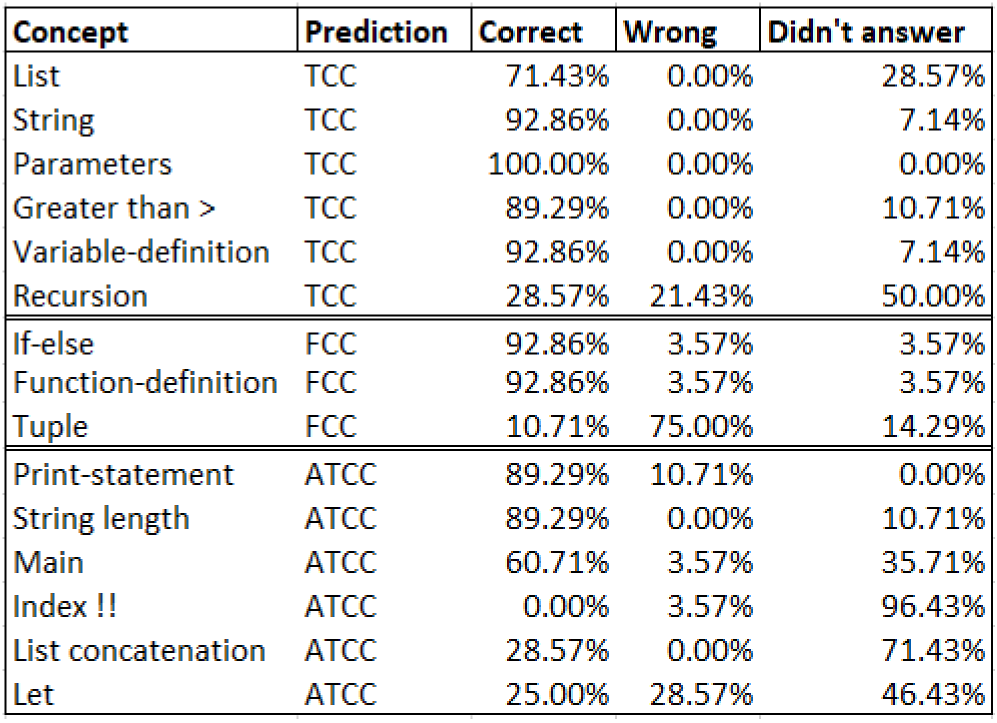
This section deals with the three quizzes. Out of these, one was created and administered by the FP lecturers at the University of Glasgow, whereas the remaining two were created and administered by the author of this thesis at the University of Oslo.

6.1.1 Haskell guess quiz

The Haskell guess quiz was held in Glasgow 11.01.2022, attracting 28 participants. Of those, 5 self-reported as having seen Haskell before, although their level of experience weren't measured. As their results followed the same patterns as the remaining students who reported no experience, their results were included with the rest.

Results

For each question, the related lines of code were broken down into their constituent concepts, which were then analyzed and noted as either 'Correct', 'Incorrect' or 'Didn't answer'.
Table 6.1 contains the results of this analysis, with answers recorded as percentages of 'Correct', 'Incorrect' and 'Didn't answer'.



Concept	Prediction	Correct	Wrong	Didn't answer
List	TCC	71.43%	0.00%	28.57%
String	TCC	92.86%	0.00%	7.14%
Parameters	TCC	100.00%	0.00%	0.00%
Greater than >	TCC	89.29%	0.00%	10.71%
Variable-definition	TCC	92.86%	0.00%	7.14%
Recursion	TCC	28.57%	21.43%	50.00%
If-else	FCC	92.86%	3.57%	3.57%
Function-definition	FCC	92.86%	3.57%	3.57%
Tuple	FCC	10.71%	75.00%	14.29%
Print-statement	ATCC	89.29%	10.71%	0.00%
String length	ATCC	89.29%	0.00%	10.71%
Main	ATCC	60.71%	3.57%	35.71%
Index !!	ATCC	0.00%	3.57%	96.43%
List concatenation	ATCC	28.57%	0.00%	71.43%
Let	ATCC	25.00%	28.57%	46.43%

Figure 6.1: Haskell guess quiz results

The results of the guess quiz are expanded upon by MPLT category below.

TCCs

The results for the TCCs were mostly as expected, with the majority of students answering correctly of the majority of students.
Notable exception was recursion, which is a more abstract concept than the rest of the examples, where 6 students answered wrong and 14 did not mention the concept at all.

FCCs

At first glance, the results regarding the FCCs seem to contradict the predictions for the guess quiz.
That so many students correctly guessed correctly for Haskell's function-definition was surprising, but the difficulty in testing the 'If-else'-concept has already been touched upon in chapter 3 'Methodology'.
Tuples appeared to catch many students unaware however, with the majority of students expecting *(w1,w2)* in *longestWord (w1,w2) =...* to simply be the parameters of *longestWord*. However, in Haskell parentheses are not required to delimit parameters, instead (w1,w2) is an example of a tuple. Only 3 out of 27 students caught this, reinforcing the prediction.

ATCCs

The ATCCs show more varied results than the other two categories.
Almost all students answered correctly for the print-statement and string-length function. However, the remaining answers had a significant portion of students who

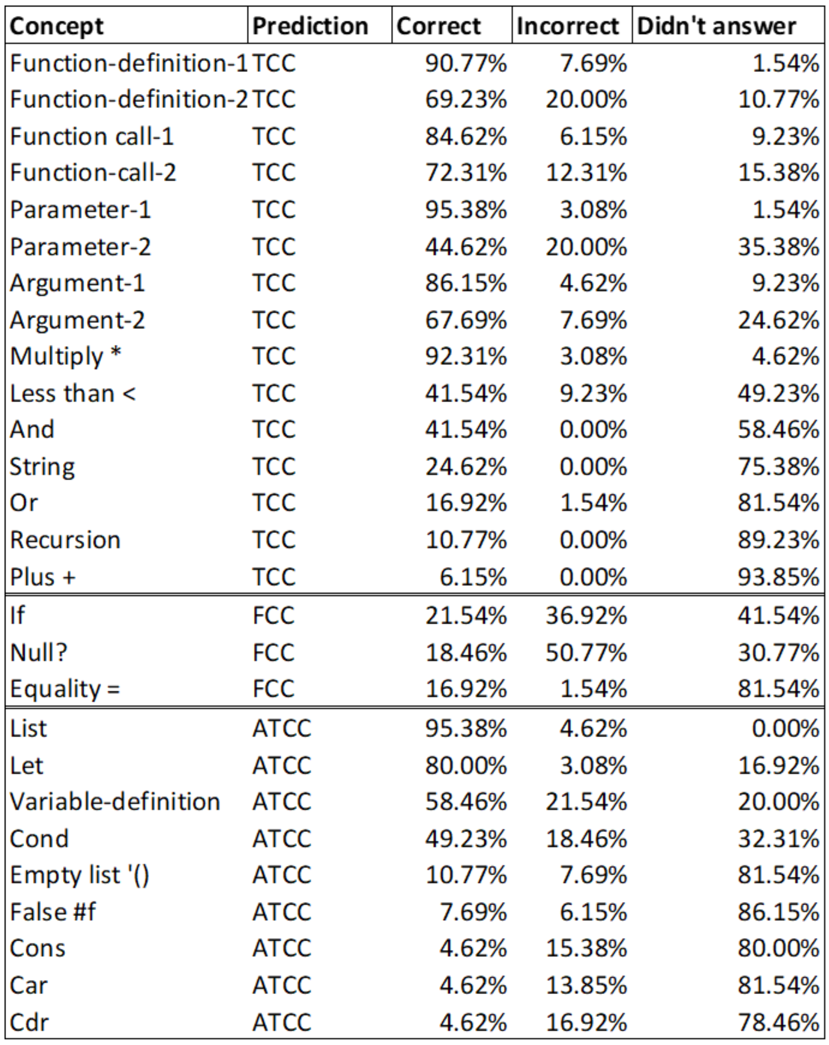
did not mention the concepts, perhaps because they did not want to admit their ignorance, or because they did not notice the concepts at all.

6.1.2 Scheme guess quiz

The Scheme guess quiz was held during the first lecture of IN2040: Functional programming at UIO, on 23.08.2022.
The quiz received answers from a total of 65 participants.

Results

For each code-example, the code was broken down into the constituent concepts, which were then analyzed and noted as 'Correct', 'Incorrect' or 'Didn't answer'.
There were two examples featuring 'Function-definition', 'Function call', 'Parameter' and 'Argument', so results for these concepts have been recorded twice.
Table 6.2 contains the results of the analysis, with percentages of answers marked as 'Correct', 'Incorrect' and 'Didn't answer'.



Concept	Prediction	Correct	Incorrect	Didn't answer
Function-definition-1	TCC	90.77%	7.69%	1.54%
Function-definition-2	TCC	69.23%	20.00%	10.77%
Function call-1	TCC	84.62%	6.15%	9.23%
Function-call-2	TCC	72.31%	12.31%	15.38%
Parameter-1	TCC	95.38%	3.08%	1.54%
Parameter-2	TCC	44.62%	20.00%	35.38%
Argument-1	TCC	86.15%	4.62%	9.23%
Argument-2	TCC	67.69%	7.69%	24.62%
Multiply *	TCC	92.31%	3.08%	4.62%
Less than <	TCC	41.54%	9.23%	49.23%
And	TCC	41.54%	0.00%	58.46%
String	TCC	24.62%	0.00%	75.38%
Or	TCC	16.92%	1.54%	81.54%
Recursion	TCC	10.77%	0.00%	89.23%
Plus +	TCC	6.15%	0.00%	93.85%
If	FCC	21.54%	36.92%	41.54%
Null?	FCC	18.46%	50.77%	30.77%
Equality =	FCC	16.92%	1.54%	81.54%
List	ATCC	95.38%	4.62%	0.00%
Let	ATCC	80.00%	3.08%	16.92%
Variable-definition	ATCC	58.46%	21.54%	20.00%
Cond	ATCC	49.23%	18.46%	32.31%
Empty list '()	ATCC	10.77%	7.69%	81.54%
False #f	ATCC	7.69%	6.15%	86.15%
Cons	ATCC	4.62%	15.38%	80.00%
Car	ATCC	4.62%	13.85%	81.54%
Cdr	ATCC	4.62%	16.92%	78.46%

Figure 6.2: Scheme guess quiz results

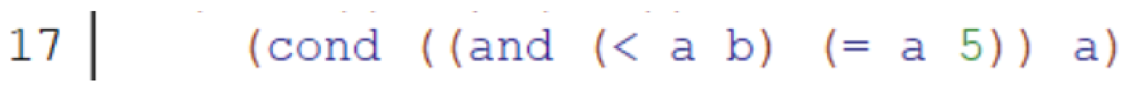
Below is a breakdown of the results based on their predicted categories.

TCCs

For the TCC-concepts, the responses were almost entirely 'Correct' or 'Didn't answer', indicating that students either correctly guessed the concept or did not see fit to mention it.
For each of the concepts that were repeated twice, the first instance of the concept garnered more 'Correct' answers than the second, indicating that students likely did not bother to mention a concept that they felt was obvious or already answered.
The only TCCs that received a notable number of 'Incorrect' answers were the second instances of 'Function-definition' and 'Parameter'. 13 students noted in their answers that they expected the code, *(define (second lst)*, to instead refer to the second index of a previously defined list.
It is interesting to note that in this case the choice of function and parameter names was interpreted by some students as an important part of the syntax.

FCCs

As expected, the FCCs received the most 'Incorrect' answers from students.
Over a third of students incorrectly assumed the if-expression would perform both following lines if the predicate proved true, and do nothing if it proved false.
Slightly more than half of the students believed *null?* would test for a *null*-object, rather than the empty list which is the case.
The equality-operator did not receive many answers however, likely because it constituted a very small part of a rather complicated code-example as shown below in figure 6.3



```
    (cond ((and (< a b) (= a 5)) a)
```

Figure 6.3: Snippet of fourth question from the UIO guess quiz

As there were so many concepts to cover in just a single line, several students seemingly focused on what they considered the 'main' part of the question, without mentioning all concepts involved.

ATCCs

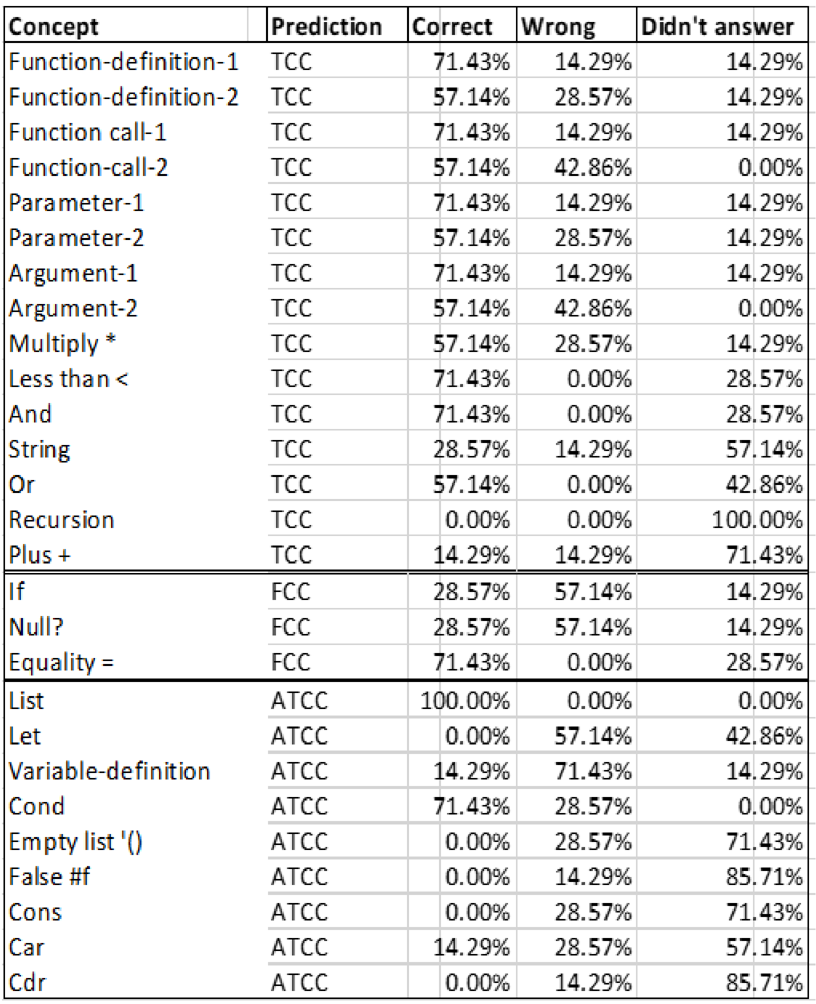
Results for this category were rather varied, with some ATCCs garnering many correct results, and some garnering almost no answers at all.
That almost every student answered correctly for 'List' was not surprising, as the difficulty in testing this concept was mentioned in the Methodology-chapter.
However, it was initially surprising how many students correctly guessed the purpose of the 'Let'-concept. It is possible the students have transferred this concept from natural languages, interpreting *(let ((a 1))* as "Let a equal 1" from English rather than any PPL.
Many students correctly related the *cond*-expression to the 'If-else if-else'-statement from Python and Java, with several noting that they made the connection due to the presence of the 'Else'-keyword.
The remaining concepts were either not mentioned at all, or students specifically stated that they had no idea what was going on in the code.

6.1.3 Novice students Scheme guess quiz

This study was intended as a small-scale exploration of novice students responses to the Scheme guess quiz, for the purpose of comparing novice guesses to the intermediate students.
The guess quiz recruited 7 participants, each having studied Python for a single semester.
By and large, the results of the novice students Scheme guess quiz mostly follows the same patterns as the Scheme guess quiz with intermediate students, with TCCs receiving more 'Correct' than 'Incorrect' answers, and FCCs receiving more 'Incorrect' than 'Correct' answers.
Excepting the *Cond*-expression and list-construct, the novice students seem less able to make accurate guesses for ATCCs than the students with prior transfer experience.

Results

The novice guess quiz was analyzed the same way as the intermediate students' quiz, as it was the same quiz in both sessions. Results from the analysis are shown in table 6.4



Concept	Prediction	Correct	Wrong	Didn't answer
Function-definition-1	TCC	71.43%	14.29%	14.29%
Function-definition-2	TCC	57.14%	28.57%	14.29%
Function call-1	TCC	71.43%	14.29%	14.29%
Function-call-2	TCC	57.14%	42.86%	0.00%
Parameter-1	TCC	71.43%	14.29%	14.29%
Parameter-2	TCC	57.14%	28.57%	14.29%
Argument-1	TCC	71.43%	14.29%	14.29%
Argument-2	TCC	57.14%	42.86%	0.00%
Multiply *	TCC	57.14%	28.57%	14.29%
Less than <	TCC	71.43%	0.00%	28.57%
And	TCC	71.43%	0.00%	28.57%
String	TCC	28.57%	14.29%	57.14%
Or	TCC	57.14%	0.00%	42.86%
Recursion	TCC	0.00%	0.00%	100.00%
Plus +	TCC	14.29%	14.29%	71.43%
If	FCC	28.57%	57.14%	14.29%
Null?	FCC	28.57%	57.14%	14.29%
Equality =	FCC	71.43%	0.00%	28.57%
List	ATCC	100.00%	0.00%	0.00%
Let	ATCC	0.00%	57.14%	42.86%
Variable-definition	ATCC	14.29%	71.43%	14.29%
Cond	ATCC	71.43%	28.57%	0.00%
Empty list '()	ATCC	0.00%	28.57%	71.43%
False #f	ATCC	0.00%	14.29%	85.71%
Cons	ATCC	0.00%	28.57%	71.43%
Car	ATCC	14.29%	28.57%	57.14%
Cdr	ATCC	0.00%	14.29%	85.71%

Figure 6.4: Scheme novice guess quiz results

A breakdown of the TCCs, FCCs and ATCCs showcased in 6.4 follows below.

TCCs

None of the TCCs received more 'Incorrect' than 'Correct' answers, and the majority of concepts got more than 50% correct.
None of the students mentioned recursion. This is not surprising as this is not a concept they should have been introduced to at this point in their education.
Curiously, one student guessed that * was Scheme's print-function, based on their expectation that there would have to be a print-function somewhere in the program. This indicates that, in addition to syntax, the students' expectations of a NPL can influence their transfer.

ATCCs

More than half the students assumed Scheme's *if*-expression would have similar semantics to Python's, assuming that nothing would happen should the predicate prove false.
As the students answering the quiz were all Norwegian, several answers seemed to conflate *null?* with the Norwegian word for zero.
The equality-operator did not fool any of the students, which they appeared to guess correctly due to the code's context of comparing various expressions.

ATCC

Excepting Scheme's list, where the difficulty in accurately testing has been explained in a previous chapter, and *cond*, where students assumed from its key-word had to do with condition, students had very few 'Correct' answers in the ATCC category.
Unlike the intermediate students, the novices did not seem to transfer *let* from natural language.
Due to how *define* was used to define a function in the first code-example, the novices assumed it would define a function as well in the second code-example as well, when it rather defined a variable.
This might indicate that they are not as flexible in their guesses, not assuming a keyword can be used for multiple things.

6.1.4 Lessons learned

Below are some notable challenges that caused trouble with the results of the guess quizzes, particularly the quizzes held at UIO.

Issues with communication

While securing permission to use the first UIO FP lecture to launch the guess quiz, the plan was apparently not communicated clearly enough.
The plan was for the lecturer to introduce the course, then have a short presentation of the guess quiz, followed by the students completing the quiz before the planned mid-lecture break.
The lecturer on the other hand believed there was only going to be the presentation, and students would then complete the quiz in their free time. This led to confusion,

as the lecturer resumed the lecture while the students were taking the quiz, and after a discussion we decided to start the break early and allow the students who wished to complete the quiz during the break.
This error in communication probably influenced the quiz negatively, as several responses showed signs of rushed answers to the last questions, likely in an effort to use the break for other purposes. Recommendations for similar endeavours in the future is to be as clear as at all possible as to what the plan is, to avoid such problems in the future.

Testing too many concepts at once

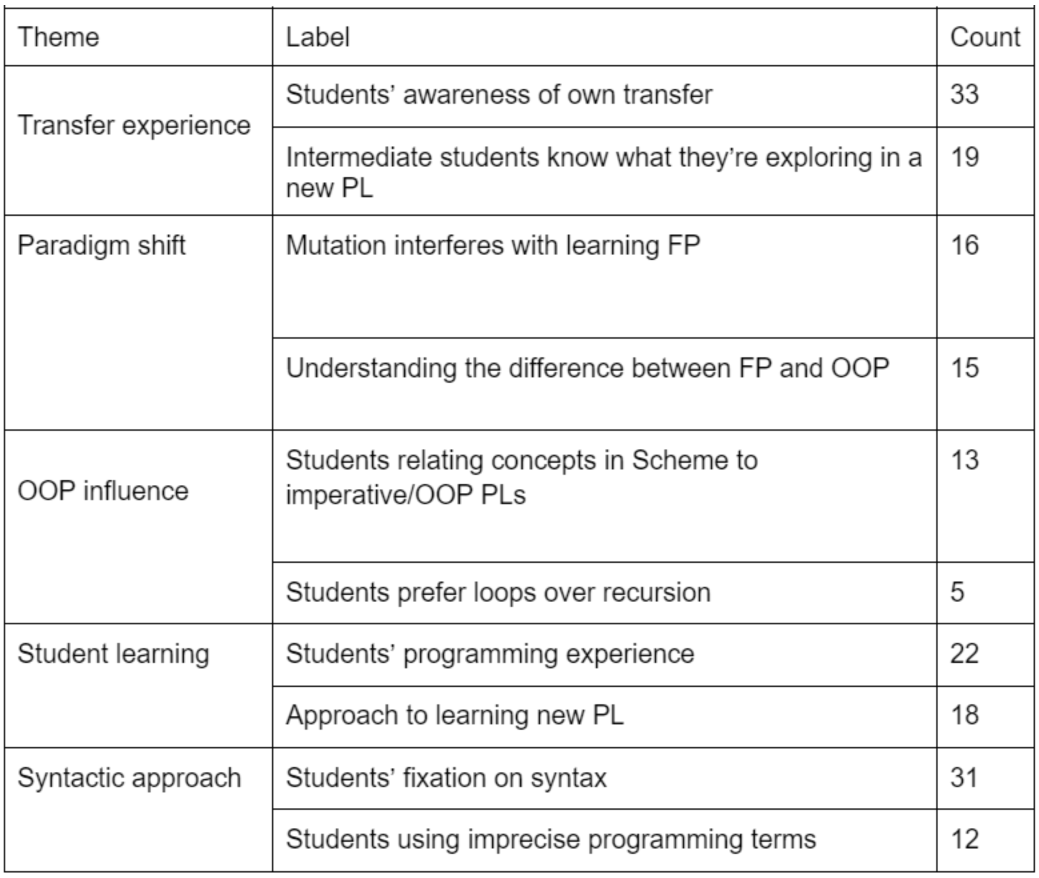
As this master's project only lasted a year and a half, there was only time to hold a guess quiz once at the start of each course. Optimally, there would be multiple chances to investigate each course over several years, allowing for incremental improvements to the process based on previous results.
Since there were no second chances, the approach chosen to construct the guess quizzes was "Test as many concepts as possible" which resulted in some questions being overloaded with 5 to 10 concepts at once. This led to several concepts not being properly covered by the students' answers, as there was simply too much going on at the same time.
It is thus recommended that in future quizzes the questions are limited to 1 or 2 concepts at a time, to better ensure students adequately cover every concept under investigation.

6.2 Interviews

This section deals with the interviews with students, teacher's assistants and lecturers.
The interviews were transcribed and subjected to thematic analysis, with the most interesting results shown in this section.

6.2.1 Student interview analysis

The interviews with students yielded many interesting answers. Analyzing the interview transcripts yielded several observations, sorted under various themes that emerged.
Some of the labels and themes found are exemplified in the below table, with a count of how many quotes associated with each label.



Theme	Label	Count
Transfer experience	Students' awareness of own transfer	33
	Intermediate students know what they're exploring in a new PL	19
Paradigm shift	Mutation interferes with learning FP	16
	Understanding the difference between FP and OOP	15
OOP influence	Students relating concepts in Scheme to imperative/OOP PLs	13
	Students prefer loops over recursion	5
Student learning	Students' programming experience	22
	Approach to learning new PL	18
Syntactic approach	Students' fixation on syntax	31
	Students using imprecise programming terms	12

Figure 6.5: Themes and labels from student thematic analysis

Of the labels shown in 6.5, a few of the more interesting ones are explained at length below.

Intermediate students know what they're exploring in a new PL

The intermediate students have seen a couple of different PLs by now, and they have started to get a rudimentary idea of what to expect in a new language in terms of concepts, tools and patterns. While they're learning they're able to see similarities to languages they know from before, and use that experience to better their learning process.
We can clearly see that P4 has seen and used function calls previously, and compares that experience favourably to how functions are called in Scheme.

> "Calls are a lot like how you always call functions, where you have the name of it and send in any arguments. And it's kinda the same when you're defining them with a name and any arguments. And you use... Do you use def in python?"[P4]

P5 is aware of conditional statements like the *if-else* construct in Java and Python, and although Scheme's *cond*-expression appears syntactically different, that doesn't prevent P5 from seeing the link after getting used to the semantics of the new concept.

> "This is what we in the world of Java and Python would call an *if-else* statement, but here it's called *cond*, or condition, in Scheme" [P5]

Mutation interferes with learning FP

While Scheme is an FPL, it includes procedures that allows for variable assignment, and thus, mutable state. P1 and P2 were interviewed before these destructive operations were taught in the course and their solutions to the coding task about calculating list length were simple and elegant, based on purely functional patterns. P3, P4 and P5 were interviewed in the weeks following the lectures on assignment, and all three immediately responded to the task by attempting to create local, mutable state with the notion of incrementing it to keep track of the list's length.
P3 eventually gave up, whereas P4 and P5 created solutions that either would not work or were grossly overcomplicated. P3 attempted to create a local variable to keep track of the list's length, incrementing it for each iteration, but along the way lost track of what scope the variable was defined in. This led to a solution where the variable was re-defined for every recursive call, and would thus always return 0.

> "I don't know about the *let*-expression, now the counter is just reset to 0 every time" [P3]

Similar to P3, P5's first instinct when starting on a solution to this relatively simple problem was to create a local variable, without really giving a reason for why regular recursion would not be sufficient.

> "We'll start by having a value, I'll write a *let*-expression" [P5]

It's surprising and a bit concerning that after months of learning the pure FP approach, the students' instincts turn to imperative solutions after only a week or two of learning mutation.

Students fixate on syntax

Scheme is a rather different language compared to what the students have seen beforehand, with a very different syntax. This crops up repeatedly during interviews where participants mention issues based in the syntax, including misunderstanding prefix-notation, being confused by key-words and especially wrangling with parentheses. There seems to be a lot of focus on getting the syntax right, to the detriment of semantics to a point where the students seem more interested in getting all their parentheses in the right place than making sure their program does what it's supposed to.
To P1, the difference between Scheme, Python and Java is mostly syntactical, and the differences in paradigms, semantics and concepts aren't touched on too much.
P1 seems to be of a somewhat higher risk of running afoul of FCCs, as they see concepts as similar as long as the syntax is similar.
This is dangerous, as the underlying semantics might be very different, without the students noticing.
Showcasing this risk, when P1 was asked about concepts that were similar between Java and Scheme, *switch-cases* were brought up as similar to *cond*-expressions solely because both start with a key-word and have surface-level syntax similarity.
When asked to explain however, P1 quickly realized the concepts actually worked rather differently.

> "Probably cond, it reminds me of Switch cases in Java, at least in the functionality."[P1]

> "Well, both start with a code word, cond in Scheme and Switch in Java. The difference is kinda that in a switch case you start with an expression and have different cases based on the expression's outcome, whereas in Scheme you also have the output on each line. Maybe not as similar as I thought really" [P1]

P2 believes the alien syntax made learning FP take longer than expected, although the syntax of Scheme is rather minimal, and the students hadn't been shown many complicated constructs at this point in the course.

> "I think I've underestimated the time it takes as the syntax is weird" [P2]

It seems as though P5 is more concerned with getting the *let*-expression right than with whether the expression fits with the context of the program.

> "I'll write a *let*-expression,that's parentheses 'l', 'e', 't', then we write a double parentheses in the let-expression, I'm not sure why, but I mean to remember that it's like that." [P5]

This seems a recurring issue, where students spend more time on getting the syntax of Scheme right, than they do learning the ideas and paradigms Scheme promotes.

Students understand the difference between FP and OO

It's interesting to observe how well the students understand the shift in paradigms they're currently doing, to see if they're aware of the differences in thinking that lies behind them and the purpose the paradigms are built around. Some students are clearly aware that there's a divide and have some ideas about how the differences work, while others seem aware that they need to think differently when working with FP, but haven't yet understood it. P1 seems to think it's all the same, just a new syntax to learn.

> "I saw a new syntax I'd have to learn, not much more really." [P1]

This is a bit worrying, it signals that difference in PLs and paradigms hasn't fully caught on yet, which may stunt P1's growth as they fail to embrace the new possibilities offered by FP.
P2 has started to see the difference between FP and OO, and views this divide as exciting. It's important to be aware that the paradigms aren't the same, and it's pleasant to see that P2 draws motivation from observing this difference.

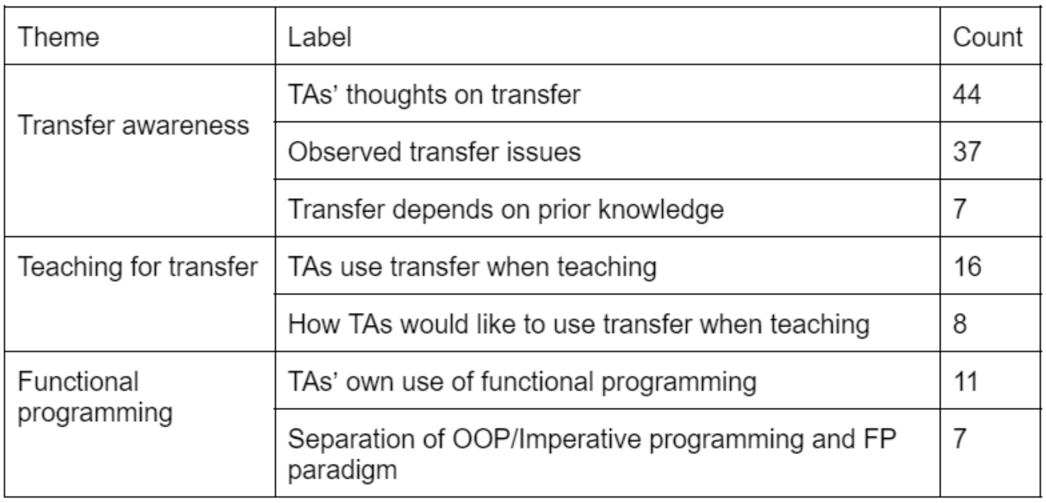
> "I thought it was exciting, you get a more defined line between functional and object-oriented programming." [P2]

P3 is aware that FP and OO are different paradigms, requiring different ways of thinking, but as of yet they aren't very aware of this difference themselves.

> "Yes, or I hear that there's a different way of thinking functional. I'm not very conscious of it, but I'm sure I've changed my way of thinking somewhat" [P3]

6.2.2 TA interview analysis

Like for the student interviews, careful analysis of the TA interviews yielded many interesting themes and labels, illustrated in the below table.



Theme	Label	Count
Transfer awareness	TAs' thoughts on transfer	44
	Observed transfer issues	37
	Transfer depends on prior knowledge	7
Teaching for transfer	TAs use transfer when teaching	16
	How TAs would like to use transfer when teaching	8
Functional programming	TAs' own use of functional programming	11
	Separation of OOP/Imperative programming and FP paradigm	7

Figure 6.6: Themes and labels from teacher's assistants' thematic analysis

We can see from 6.6 that the TAs had a lot to say about transfer and how to teach for it. Some of what they said will be expanded upon below.

TAs' thoughts on transfer

While PL transfer was not the first thing that came to any of the TAs' minds when asked, they all had opinions on PL transfer. They had ideas about what concepts it works for, like and and or, lists recursion and similar.
They also had differing opinions on the usefulness of PL transfer. TA1 advised against relying on transfer too much for fear of becoming dependent on a PPL to understand the NPL, TA2 considered transferable concepts to be amusing trivia, whereas TA3 thought syntax transfer is doomed to fail.
Each of the TA's had a different idea of PL transfer when the idea was brought up, with TA2 relating PL transfer to teaching PLs in general.

> "First thing that comes to mind is transferring knowledge from teacher to student" [TA2]

TA1 is somewhat skeptical to the idea of relying on transfer, particularly aware of the danger that leaning too much on transfer can make it hard for students to de-couple their understanding of the NPL from the PPL they transferred from.

> "I also feel that you should try to detach the students from that thinking. Like you can do that in the start of the semester, but if you go the whole semester saying "You can do that like this in Python" or "You can do it like this in Java", then the students don't get that independent thinking of functional programming" [TA1]

TA3 has observed issues with students trying to learn a new PL by transferring syntax, and as such has grown distrustful of the idea.

While this approach to transfer avoids the danger of FCCs, a purely conceptual approach to transfer risks losing out on a lot of the benefits from TTCs.

> "Sometimes people try to transfer syntax, that often doesn't go so well, like trying to transfer syntax from Scheme to E-Lisp, Java to C, from C to C#. It doesn't always match perfectly, there will always be some small changes" [TA3]

Syntax transfer does indeed come with a set of pitfalls, which makes TA3 consider the potential positive impacts not worth the risks.

TAs use transfer when teaching

The TAs don't appear to be very conscious of using transfer when they're teaching, or if they were at one point aware of it, they've forgotten to use it as the semester progressed. However, they can give examples of situations when they've compared concepts students knew from PPLs as an aide to explain concepts in Scheme, facilitating transfer.
When asked how they would plan if they were to teach for transfer, all three said they'd give examples of code in both the NPL and a PPL side-by-side, to make the parallels explicit. TA1 had ideas about transfer being a useful concept for teaching, and tried using it to teach students initially, but eventually forgot the approach as the course went on.

> "I should, and I feel, like especially at the beginning I go like "this is similar to this thing in Python" "This is like that" And then I give some tasks like "translate this Python program to Scheme" and the other way around, translating Scheme to Python, but that I kind of forget it over the course of the semester" [TA1]

TA2's use of transfer doesn't strictly relate to teaching Scheme, but rather how concepts from FP can be useful outside of the course as a way of motivating the students to take the course seriously. By explaining concepts that can benefit the students in the PLs they use in their other courses, TA2 aids them in looking at FP beyond just the boundaries of the course.

> "I've actually used it a bit as a group teacher. I've mentioned stuff like "See map and filter? You can actually come across these outside the course as well!" Recommend they try that a bit" [TA2]

TA3 is no stranger to transfer, using concepts from many different courses to explain the inner workings of Scheme. However, the examples used are typically very abstract, usually not getting into the nitty-gritty details of the syntax and semantics of the concepts in different PLs. When specifically asked how PL transfer could be better facilitated, TA3 proposed showing examples of concepts in several PLs side-by-side, to better showcase the differences and similarities between them.

> "I would point out functional principles in languages they have learned, so if they've learned something in Python, they've learned something in Java, I mean Java has streams, I'm not sure if they have something similar in Python. . . But I would try to use an example in the old language and the new language and show them side by side, I think that's what I would do" [TA3]

Despite varying degrees of transfer-based teaching approaches, all three TAs suggested this approach to PL transfer during their interviews. By showing code-examples side-by-side, it's simple to drive home to the students that they actually have experience with an ATCC, despite how novel it might appear, and the TAs can correct students' confusion regarding various FCCs that crop up.

Observed transfer issues

The TAs are positioned well to closely observe the students, their progress and the challenges they face. They've all noticed issues that can be attributed to transfer, where students are used to solving problems differently, and bring that way of thinking into Scheme.
All three noticed examples of FCCs, with students having occasional when they try to transfer the syntax of a concept they knew from a PPL without considering the ramifications of semantics.
A big, recurring issue brought up by TA1 and TA3 is mutation, where the students' OO and imperative backgrounds shine through and they get stuck trying to use destructive operations as much as possible. This often leads to confusion and failure, and can cause students to miss out on learning true functional programming.
Especially prevalent in the first few assignments, the students start programming in Scheme expecting to write code the same way they're used to in Python and Java, but are rudely disabused when *(+ a 1)* doesn't actually change *a* in any way.

> "In Java you write like *a += b*, and then you change the variable *a*, and I see that all the time in the first assignments that you get some assignment when you're supposed to take in a number and return a new number and they try to destructively change some variables outside the procedure" [TA1]

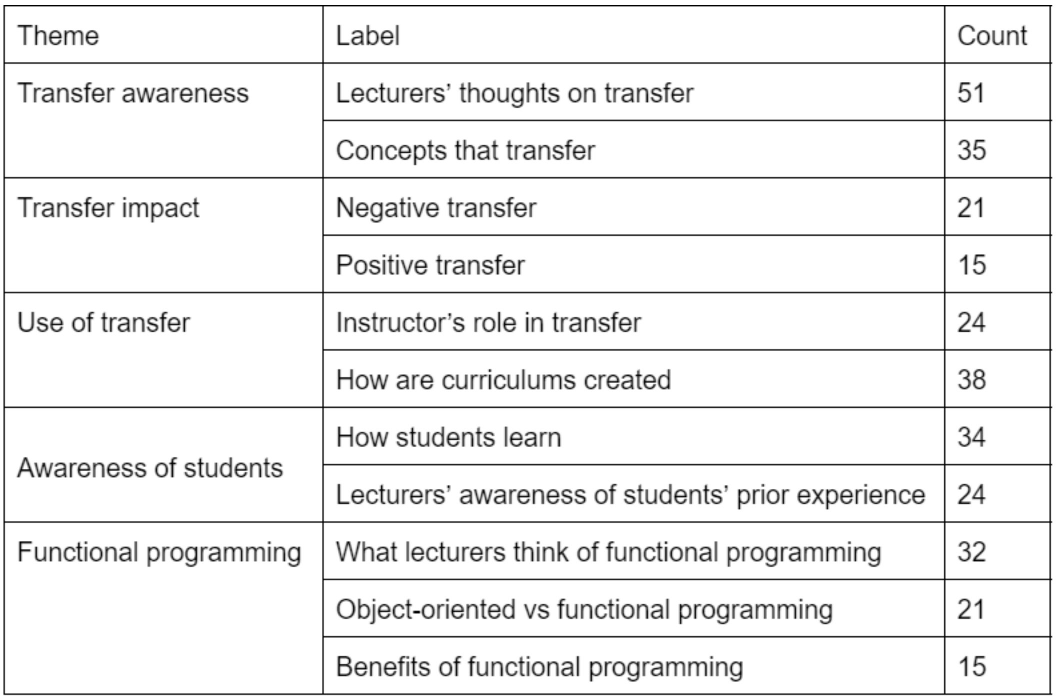
We can also see from TA3's observations the effect mutability has on learning FP, as students quickly latch onto destructive operations in Scheme as soon as they've learned how.

> "I think often it goes very abruptly from an elegant function, which rapidly gets very ugly when you add mutability. It just doesn't look as pretty because you always need extra stuff, extra calls to set! on very many things, and I think that is a thing that makes people make things harder for themselves. They force themselves to keep going in this way they've just been taught, instead of holding on to the other way which we want to do in Scheme" [TA3]

Students are very familiar with imperative procedures that mutate variables from their PPLs, and when they learn to *set!* variables in Scheme they quickly latch onto their old habits, even when the functional approach is simpler.

6.2.3 Lecturer analysis

Unsurprisingly, the lecturers had a lot to say on a wide range of topics relating to FP, teaching and transfer. There was a large effort associated with narrowing it all down into a few themes and labels, exemplified in the table below.



Theme	Label	Count
Transfer awareness	Lecturers' thoughts on transfer	51
	Concepts that transfer	35
Transfer impact	Negative transfer	21
	Positive transfer	15
Use of transfer	Instructor's role in transfer	24
	How are curriculums created	38
Awareness of students	How students learn	34
	Lecturers' awareness of students' prior experience	24
Functional programming	What lecturers think of functional programming	32
	Object-oriented vs functional programming	21
	Benefits of functional programming	15

Figure 6.7: Themes and labels from lecturer thematic analysis

It has been challenging to narrow down the labels from figure 6.7, but some of the most relevant observations have been collected and expanded upon below. L1 and L2 are the lecturers from Glasgow, while L3 and L4 are the UIO lecturers.

Benefits of transfer

The lecturers have a mostly positive view of transfer, considering it healthy for students to view concepts and problems from multiple differing perspectives. While there are issues to address with certain instances, transfer overall is a good thing in their minds.
Transfer helps students learn a NPL faster, being able to lean on what they knew before, and can also strengthen their knowledge of the PPL they are transferring from.
Transfer can help mitigate students' feeling of being overwhelmed by a NPL, by throwing students a lifeline to hang on to from the PPLs they're already comfortable with. L1 is used to taking advantage of transfer when learning new PLs, using familiar concepts in multiple PPLs to quickly transfer knowledge of concepts from multiple PPLs.

> "For me it works really well because I go to a new language and then I think I say, oh yeah, that. I know that from language X. And that's just like in language Y, only a little bit different." [L1]

A perhaps infamous concept in Haskell is the monad, and L2 notes that many students arrive in class fearful of the concept, as they've read "horror stories" online. To address this fear, L2 compares monads to concepts the students are already aware of, to explain that they're not as alien or terrifying as they appear at first glance.

> "With that concept in particular for Haskell, there's always a lot of, I think fear on the part of students, because they've heard “Ohh, monads

> are scary!" and they see all this stuff online and so I think there's some level of trepidation on part of students, so to be able to give them a sense that this isn't really anything magic, it's really well principled and , you know, you might see some aspects of it in other languages, I think that's helpful" [L2]

Transfer doesn't just work one way, as described in Bower et al[11]. There is both proactive facilitation, when learning a new PL is aided by knowing an old one, and retroactive facilitation, when learning a new PL strengthens your knowledge of the old PL.
L3 is of the same opinion, viewing PL learning as a chance to not only broaden their horizons by learning something new, but also improving their knowledge of PLs they already knew.

> "Yeah I think that there's some kind of two-way enrichment, when you learn" [L3]

L4 is very positive to transfer, particularly to teaching students to view concepts from multiple angels, decoupled from the context of a specific PL.

> "I'm fully in favor of that, because it's only by seeing from different angles, after a while you get "grown-up" as a programmer, not learning a particular definition, you see the concept somehow." [L4]

By looking beyond a given PL, students become more aware of the concepts themselves, allowing them reason about a program solution without tying themselves too tightly to the syntax of their favored PL.

Lecturer's awareness of students' prior knowledge

In order to teach for transfer, the lecturers need to have some awareness of what PLs and concepts their students are familiar with.
L1 and L2 both have decent knowledge of their students' capabilities, although this is in large part due to their extensive involvement in teaching the same students at lower levels. L2 teaches level 1 and L1 teaches level 2 and 3, so together they have a pretty solid understanding of where their students are in terms of their prior knowledge.
L4 just recently took over the FP course at UIO, and doesn't yet have a solid picture of what his students know, other than a vague idea that they've seen Python or Java. Being aware of what students know helps immensely when teaching, as it helps the lecturer pick examples to explain. Other than being aware of what students know, it is also very important to be aware of what students **don't** know, as it is easy for lecturers to assume a concept is obvious despite students never having seen it before. L1 mentions how his previous lack of knowledge about students' experiences led to some misconceptions about what students would have learned previously, like assuming they had experience with OO and FP from their first year.
These kinds of misconceptions about students' experience can lead to issues when teaching for transfer, as examples of expected prior knowledge will simply confuse students who do not actually posses that knowledge.
L1 also talked about how this seems to be a common problem with lecturers, pointing

to a Glasgow course that teaches C, where the lecturers erroneously expect the students to have experience reasoning about memory management before taking their course.

> "So I think now I'm quite aware of what our students know. Probably 10 years ago, I wasn't that aware. For example, I didn't realize that we didn't teach OO and anything functional in the first year of Python course." [L1]

Similar to L1, L2 believes the efficacy of transfer depends on having a solid awareness of your students' level of experience. When you know what concepts your students are familiar with, you can use those concepts as a tool to teach similar concepts in the NPL, bridging syntactic differences and pointing out differences or similarities in semantics.

> "I think a lot depends on, you know, whether you can be sure that all your students have a similar background, and whether you're aware of what that is" [L2]

As L4 has only recently started teaching the FP course at UIO, the level of awareness about students' prior knowledge is much lower than that of L1 and L2.
This leads to issues as their experiences are consistently over- and underestimated, leading to tasks that are at times unnecessarily complicated, and examples that fall flat for the students, for instance L4 would occasionally attempt to use examples from a compiler course they also taught, but eventually it turned out that none of the students had actually taken the course in question.

> "Of course I have assumptions but I basically don't know. I mean there's the introduction to programming, I think in the old days with Java now it's Python or Java or something like that, but, I mean I'm sometimes surprised"

The role of the instructor in guiding transfer

For this theme, opinions seem a bit split along rather natural lines. L3 doesn't teach FP, so his answers are mostly anecdotes from his days as a TA, and L4 has not yet had time to start thinking about transfer-based teaching and so had little to say on the topic.
L1 and L2 however, have been teaching FP for many years now, and have some firm ideas. They seem to believe quite a lot in the importance of the teacher in the transfer process, as someone who guides students' transfer along the correct lines and corrects misunderstandings as they crop up. They've tried several different approaches to teaching for transfer, and believe transfer to be a positive thing as long as the teacher does a good job of leading the students in the right direction.
When teaching for transfer, the approaches favored seem to be using old concepts in a PPL to give examples as a way to explain concepts in the NPL, sometimes showing code-examples side-by-side, and being explicit about when familiar ways of thinking will work and when they will not. L1 relishes using examples from multiple PLs as a way to show students that CS education is less about what PLs you've learned, but rather what concepts you know.

> "I like to show how you can do the same thing in many different languages, cause it's about concepts, it's not about languages. So it's nice to know you can do maps and folds in Python as well as in Haskell" [L1]

L2 points out that students will instinctively transfer concepts regardless of teaching, and so thinks lecturers should work to take advantage of that natural transfer. Since transfer cannot be stopped, then like a river it should be guided. Lecturers can and should help facilitate students' transfer, aiming it in the proper direction and away from misconceptions.

> "I think it's natural and instinctive for students, and I think we should take advantage of it in our teaching, try and facilitate that transfer and sort of make it explicit in our teaching." [L2]

L4 worries that students might not appreciate transfer, seeking to avoid unnecessary examples that are not strictly curriculum. This is not a totally unfounded worry, as both L1 and L2 has pointed out that some students complain when given examples of multiple PLs they do not know.
However, they have not mentioned any complaints regarding examples using PLs that students **are** familiar with, so using PLs familiar to the students should help avoid the confusion the complaining students mentioned.
As long as the examples are given in PLs the students are familiar with, severe issues should not arise, and students who do not wish to learn anything outside of the curriculum are free to ignore these at leisure.

> "The only danger is how to make it clear that's it not curriculum, it's only meant as helpful, but on the other hand to avoid that the students simply ignore it like 'I don't need to know that, how it looks in Python, I won't read it, I'll focus on what is required in the end'" [L4]

The MPLT and concepts that transfer

The MPLT can be used to predict the transfer of concepts from one PL to another based on their syntactic and semantic similarity. While L2 actually discussed the model with its original creator in regards to teaching Haskell, the lecturers had yet not attempted to employ the model in their teaching.
Nevertheless, they all had notions of concepts that transferred either correctly or incorrectly due to the interrelation between syntax and semantics. L1 has mentioned a concept that is a clear example of a TCC: mathematical operators and their relation to integer and real number arithmetic.
The vast majority of PLs use the same operators for mathematical operations like addition, subtraction, division and similar, and in most PLs, these operators also work the same way regardless of whether the number in question is an integer, a float or any other number type.
These concepts are both syntactically and semantically identical across Python, Java, Scheme, Haskell and most other PLs, making them rather good examples of TCCs.

> "The basic assumption that the plus sign can be used for integer operations as well as for real operations and so on. That's pretty much in any programming language. There's very few programming languages where

> you need different plus and multiply signs for different number types. But that's clear transfer. You're assuming it will work like that. And it does, because it would be crazy if it didn't." [L1]

The MPLT predicts that Haskell's *if*-expression is an FCC, as the syntax looks very similar to if-statements in Java and Python, while its semantics is closer to the ternary-operator.
L2 has apparently noticed this concept as an FCC when transferring from C as well, and with the awareness of the confusion FCCs can cause, has chosen to make explicit the semantic difference from the if-statement students are used to.

> "For instance, the ternary-operator in C, you know, question-mark colon (*? :*), that's really the same as the if-then-else in Haskell, and it's important for them to realize that if-then-else in Haskell is an expression, which is different to an if-then-else in C or Python which is a statement, so I have to say you know, like 'This looks like a C statement, but actually it's the C ternary-expression' "

It is noteworthy that making this comparison explicit does not require much effort on the part of the lecturer, with L2 settling on a simple approach like "You can see that concept *A* looks like concept *B*, but actually you'll find that it works like concept *C*".
Teaching for transfer does not need to be an arduous, time-consuming process, it simply requires a bit of knowledge, some planning and a short explanation. L3 did not previously have knowledge of the MPLT before the interview, but when presented with it found its ideas made sense.

> "Yeah I like it, I like it because it's like, the first category has a lot to do with why, like C# and Java is easy, it's a simple transition because it both looks similar and the general shape of the code makes it all feel a bit like home"[L3]

The model not only aids in teaching for transfer, but also helps explaining why PLs that extensively shares syntax and semantics are simple to transfer between even for those unaware of how transfer works.

Challenges with teaching for transfer

While the lecturers praise transfer-based learning as a useful approach, it is not without its own challenges.
As L4 has mentioned, it can be difficult to find the correct balance between clarifying that a given example is not strictly curriculum, and ensuring the students take the examples seriously enough to avoid them simply ignoring the example because it is not strictly curriculum.
L1 has had experience teaching more than one PL in a course, to promote comparison between them to aid in students' learning, but this left the students feeling overwhelmed and confused, and so the attempt had to be abandoned.

> "I taught two languages, one was Haskell and the other was Lifescript" "It is basically a functional language, but dynamically typed and compiles to Javascript. So it looks really Haskell-ish, but without the static typing. And I used this to compare and contrast approaches. But the students, even though it was so easy you could very easily just copy and paste the code almost, they still didn't like it. They thought it was confusing, so I learnt from that that it's better not to do that, although personally I like this." [L1]

When teaching Haskell at Glasgow, L1 and L2 can plan their transfer approach ahead of time, based on their extensive knowledge of what concepts their students should be aware of.
However, they have also created a Massive Open Online Course (MOOC) teaching Haskell as well, where it is not possible to take such a measured approach to transfer as the students come from so many different backgrounds. Instead, they attempt more of a "throw transfer at the wall and see what sticks" approach, where they show examples from many different PLs and hope the students find at least some of the examples helpful.
While this approach is helpful to some students, others have complained that they do not recognize any of the example PLs, and as such feel they have to learn multiple PLs just to keep up with the course.

> "So I guess it's a more random approach where we try to cover a set of different languages and hope that something is recognized by the student. That doesn't work for everyone to be honest, and some people complain about it, they don't like that because we haven't found something they're familiar with, and then we've confused them because they have to look at two languages rather than one, and that's a difficulty with that particular course." [L2]

The challenges present in L1's and L2's experiences seem to stem from students' reluctance to see examples from PLs they are not familiar with.
When they are shown PLs or examples from PLs they do not know, this does not immediately benefit, and they do not seem to find the extra effort to benefit from these examples worth it.
While it might be argued that this is "laziness" on the part of the students, it is important to keep in mind that the main point of using examples in the first place is to facilitate learning, and if the examples instead increase the effort the students need to learn, then the example is not useful.
To help facilitate students' learning through transfer, more suitable examples must be chosen, and as the complaints appear to arise from students not knowing the PLs used in examples, it is vitally important that lecturers pay attention to students' prior experiences.

Chapter 7

Discussion

In this chapter the results of the thesis will be discussed in the context of proving the hypothesis of this thesis:

> The Model for Programming Language Transfer transfer categories are in evidence when intermediate students from an Object-Oriented background transfer to Functional Programming and exploring this transfer with students and their teachers could improve the process.

The discussion will consider the gathered data, using the stated research questions as a tool to shape the discussion into relevant, comprehensible sections.

7.1 RQ1: What outcomes would be predicted by the MPLT for an intermediate student's first encounter with FP

The topic of what predictions the MPLT can make has been extensively discussed in chapter 4, Predicting Transfer.

7.2 RQ2: How does the intermediate students' experience transferring to FP match the predictions of the MPLT

This section looks at the results from the guess quizzes and interviews to discuss the validity of the MPLT's predictions.
The discussion is divided into smaller discussions regarding the prediction categories: TCC, FCC and ATCC, as well as discussing students fixation with syntax.

7.2.1 Accuracy of predictions

In order to validate the accuracy of the predictions made with the MPLT, the results of the guess quizzes will be used. These quizzes were made using the MPLT's predictions of how students would transfer PL concepts, in an attempt to ascertain whether the predictions matched with reality.

According to the MPLT, students are likely to correctly guess how TCCs work in the NPL, while they are likely to have misconceptions about FCCs, believing them to be TCCs based on syntax. ATCCs are expected to be perceived as novel by the students, and as such there is expected to be little to no transfer.

True Carryover Concepts

If the MPLT's predictions concerning TCCs were correct, we would see students answering mostly correct for these concepts, with few incorrect answers.
The results from the guess quizzes seem to validate these predictions, as the vast majority of students that answered, answered correctly for most of these concepts.
The interviews with students compound these findings, as the students themselves mentioned experiences where the similarity of concepts simplified transfer. The concepts that garnered the most incorrect answers, at 20% each, were *function-definition-2* and *parameter-2* from the UIO guess quiz. However, these misconceptions resulted from students incorrectly drawing context from their names, *(second lst)*, as having to do with either a second list or the second index of a list.
By and large, we can see the MPLT's predictions regarding TCC as validated. Students' transfer of TCCs are positively impacted by the syntactic and semantic similarity.

False Carryover Concepts

According to the MPLT's predictions, we should see many students answering incorrectly about the FCCs, as their similar syntax make the students guess incorrectly about the concepts' semantics.
These predictions seem to match with the quiz results, as many of the FCCs have more incorrect answers than correct.
There are several exceptions where students correctly guessed the FCC, like Haskell's *if*-expression. The difficulty in accurately testing this concept was discussed in chapter 4. A danger with FCCs is that students themselves do not necessarily realize they have the wrong idea about a concept. The TAs realize however, and could tell of many observations with students getting a FCC wrong due to transferring syntax.
Looking at these results, it appears as though the MPLT's predictions about FCCs are validated. There are cases where students still understand the concepts, and when they get the wrong idea they can occasionally self-correct, but the negative impact of FCCs on transfer is still something a teacher should keep an eye on and take efforts to address before the students have time to misunderstand them.

Abstract True Carryover Concepts

As the ATCCs share similar semantics, but not syntax to the students' PPLs, the MPLT predicts that students will perceive ATCCs as novel, limiting their ability to transfer the concept.
The results of the guess quiz show more variation than the other categories, with both correct and incorrect guesses. There are also a lot of ATCCs that were not mentioned by the students, as they either did not notice them or did not want to venture a guess as to how they worked.
For several of the ATCCs introduced, like *cons*, *car* and *cdr*, many students explicitly

stated that they had no clue what was going on.
As the students appeared to generally have trouble with the ATCCs, although not as much trouble as expected, the MPLT's predictions about students perceiving ATCCs as novel can be seen as valid.

7.2.2 Students fixate on syntax

The basis of the MPLT is the idea that students transfer PLs by looking at syntax first and making assumptions about the semantics based on similarity of the syntax in the NPL compared to one or more PPLs. The quizzes and interviews show that this does indeed happen. The results of the quizzes show that students guess correctly or incorrectly about the semantics of a concept based on their perceived familiarity of its syntax, roughly along the same lines predicted by the MPLT.
Students are more likely to guess correctly for the TCCs, incorrectly for the FCCs, and less likely to make correct guesses about the ATCCs.
The interviews with students also show this trend of syntax fixation, with the students spending considerably more time talking about specifics of syntax than they do the semantics involved. During the interview programming tasks, they also showed that they paid more attention to spelling out the syntax of a test program correctly than they did worrying about whether the program would actually do what it was supposed to, like P5 carefully placing each letter and parentheses of a *let*-expression without taking the time to consider whether a *let*-expression was really helpful to their solution.
Syntax is obviously important to students, and in their minds it appears difficult to separate what the syntax of a concept seems to say, from what it actually does. Even at an intermediate level where the students should have gone far beyond just learning the syntax of PLs, delving into various data structures, algorithms and programming practices, the intermediate students still build programs bottom-up like Scholtz et al[25] reported novices do, rather than plan their approach ahead of time like the experts.[26].

7.2.3 RQ2 discussion summary

According to the MPLT, when the students see a concept in a new language, they will make connections based on the syntax of the concept. If the syntax is similar to what they are used to from PPLs, they will assume the semantics work similarly as well.
For TCCs this affects transfer positively, and we can see from the guess quizzes that students are very accurate when guessing the workings of these concepts, while for FCCs the transfer is negative and so we see the students make many incorrect assumptions instead. The ATCCs do not share syntactic similarity, so students often assume they are novel despite actually having experience with similar semantics. The results are more varied for students guessing ATCCs, but they are not as accurate as when guessing a TCC. The intermediate students were however somewhat more accurate when guessing ATCCs than the novices, which will be discussed in the next section.
The guess quizzes and interviews show that students do indeed fixate on the syntax of concepts, letting the syntax color their assumptions about a given concept's

semantics, and paired with the guess quiz findings it appears that the predictions made with the MPLT about transfer to the FPLs Scheme and Haskell are validated.

7.3 RQ3: What is the effect of students with prior programming transfer experience on the predictions/effectiveness/usefulness of the MPLT by comparison with novices transferring?

To validate the MPLT, Tshukudu's [1] studies focused on how novices transferred from their first PL to a second.
However, the FP courses at UIO and Glasgow are available to students between their second to fourth year of university, which means the students will already have at least one year of programming studies, teaching at least two PLs.
As the students already have experience transferring from one PL to another, it is relevant to study whether and how this experience impacts the predictions of the MPLT, to see whether the teaching of intermediate students must be addressed differently compared to pure novices.

7.3.1 Intermediate students' transfer compared to novice student transfer

This thesis administered the same Scheme guess quiz to both intermediate students and novice students. The reason for having novice students do the quiz was to have the novices serve as a standard by which to measure the intermediate students up against, to see if the intermediate students could make more connections than the novices, with more experience programming, knowing more concepts and having made the jump from one PL to the next several times before.

Variable bindings vs function definition

For the first task of the Scheme guess quiz, the students were shown the keyword *define* being used to define a function. Both the novices and intermediate students scored highly at correctly identifying this example. However, for the next task the students were shown *define* used to bind a variable. For this example, 71.43% of the novices still believed *define* would define a function, only a single student believed that they were seeing a variable binding. On the other hand, only 21.54% of the intermediate students made the same mistake, with 58.46% correctly identifying it as a variable. This shows that more of the intermediate students than novices are flexible enough in their assumptions to allow for keywords that can have different meanings in different contexts.
Prior transfer experience has shown the intermediate students that a NPL can have very different syntax and semantics from what they're used to, and while their transfer is by no means perfect they show more awareness that other PLs might behave differently, and are capable of holding multiple models of what a given keyword signifies in different settings.

Intermediate students know more concepts

The MPLT predicts that students will not be very accurate in guessing the semantics of ATCC, given the concepts' apparently novel syntax. While this prediction mostly holds true, the apparent novelty from the ATCCs seem to impact the novice students to a larger degree than the intermediate students.
While the intermediate students show low scores for ATCCs like the empty list, *cons*, *car* and *cdr*, the novice students show no correct answers for all but one of these. While not directly comparable, the Haskell quiz also demonstrates that there are at least some students who correctly guess most of the ATCCs they were presented with.
The data should be taken with a grain of salt however, as the intermediate student quizzes had much larger sample sizes than the novice quiz, but the intermediate students prior experience does facilitate transfer due to one important factor: knowledge.
The intermediate students have shown a more flexible approach to concepts and their syntax, but they also benefit from knowing more concepts that transfer.
None of the novice students made any guesses regarding recursion. This is not unexpected, as the novice students would not have been taught about recursion in their first semester. It was actually rather impressive that one novice student managed to correctly guess that *car* was a function for retrieving the head of a list, because unlike the intermediate students they would not have been taught about linked lists. This concept was only predicted as an ATCC for the intermediate students due to their known prior experience with the concept.
The knowledge that comes from prior transfer experience, and more programming experience in general, greatly benefits intermediate students' transfer for the simple reason that they have more concepts and PPLs to draw from when transferring knowledge to a NPL. When predicting transfer with the MPLT, this means that there are more concepts to account for when predicting TCCs, FCCs and ATCCs, seeing as concepts such as recursion and linked-lists could not be transferred to an NPL by novices who do not know how they work in their PPL.

7.3.2 RQ3 discussion summary

Contrary to pure novices, intermediate students already have some experience with transferring PLs, which affects their transfer to a NPL.
The impact of prior transfer experience seems positive, as the intermediate students show more flexibility in their approach to a new concept, being readily prepared to consider that the same concept can have multiple different meanings, which helps them transfer more ATCCs. In addition to this, they simply having more knowledge from their multiple PLs, which give them more prior knowledge to draw connections from.

7.4 RQ4: How does having an OO background impact students' experience transferring to FP?

This section looks at how students are influenced by their OO backgrounds when learning FP, primarily looking at the students learning Scheme at UIO.

A big issue interfering with students' learning FP was mutation, which will be discussed extensively below, as well as students' preference of loops over recursion.

7.4.1 Interference of mutation on FP transfer

In pure FP, mutable state is not used as an important part of the FP paradigm is to avoid shared state and side-effects. The students are however used to OO, where shared, mutable state is the norm.
A recurring issue brought up in the interviews with students, TAs and lecturers is how the introduction of mutable state to an FP language causes the OO background students to fixate on mutable concepts to the exclusion of pure FP concepts.
This seems to be an issue in particular for the UIO course, IN2040: Functional Programming, which is largely based on the book Structure and Interpretation of Computer Programs[27]. The book focuses on using Scheme to illustrate various data structures, rather than focusing solely on pure FP.

Students experience issues with mutation

When interviewing the students learning Scheme at UIO, a curious pattern emerged as all five students were asked to write a simple program for calculating the length of a list. The first two students were interviewed before mutation had been introduced in lectures, and they both provided simple, purely functional solutions. The last three students however, were interviewed after mutation had been introduced, and all three of them immediately attempted to introduce mutable state to their solutions, leading to solutions that didn't work, were needlessly overcomplicated or they gave up halfway through.
After at least two months of learning pure FP, the three students had at most one or two weeks introducing mutation, but immediately made the switch to using mutable operations as their first attempt. This is likely due to their fragile knowledge from two months of FP learning competing with at least one year of OO experience.

Teacher's assistants observe issues with mutation

Two of the three TAs interviewed from the UIO FP course also mentioned issues with students fixating on mutation as soon as they learn it. They observe the issue both in their lessons and when correcting the students' assignments, and note how the students' code grows more complicated and confusing, with students forcing themselves to use *set!* to destructively change state to the detriment of the pure FP the TAs want them to, like TA3 mentions:

> "It just doesn't look as pretty because you always need extra stuff, extra calls to set! on very many things, and I think that is a thing that makes people make things harder for themselves" [TA3]

This is exactly what was observed with the students solving the list-length task in their interviews, which shows that the three interviewed students who tried introducing mutation are not abnormal, nor a minority. The TAs observe the issue of mutation interfering with students' learning FP regularly, severely complicating the code students write and negatively affects their learning.

Lecturers have experienced and observed issues with mutation

The lecturers have also mentioned issues with mutation when teaching FP, issues that go beyond the UIO course and showcasing that the issue isn't isolated to a single course teaching a single FPL.
The lecturers even had personal experience with mutation interfering with their own learning of FP in their earlier years, L1 having had their experience learning ML ruined by mutable state they had experience with from Basic, and L3 having run afoul of the issues with *set!* in Scheme.
L3 also mentioned observing both students and colleagues running afoul of the same issues in both Scheme and Elm, noting that both complained of disillusionment with FP when their code was obviously not written after the functional paradigm.
Based on the interviews with the lecturers, issues with introducing mutation to someone learning FP is not only limited to students at UIO learning Scheme, but to programmers in general transferring to the FP paradigm.

7.4.2 Students prefer loops over recursion

One of the tasks students were asked to perform during the student interviews was to write a simple piece of code to calculate the length of a list in Scheme.
Regardless of whether they succeeded or not, they were then asked to explain how they would have done the same task in Java or Python.
Without fail, every single one of them said they would use a loop, rather than recursion. That in itself is not unsurprising, as they are used to solving problems with loops in those PLs. However, not only did the students explain that they wanted to use loops, they were all explicit, some borderline gleeful that they would not use recursion.

> "Oh, I would just use a ... for int i = 0, i less than length of list, where you increment the length for each time..." **Interviewer: "You would use a loop?"** "Yes! No recursion!" [P4]

Having had at least a year, in multiple PLs, where loops are the go-to approach for problems that require repeating code, switching to recursion requires a considerable mind shift.
Given their apparent dependency on mutation, one has to wonder if the students would use recursion at all if they learned how to make loops in Scheme.

7.4.3 RQ4 discussion summary

The students involved in this study have extensive experience with OO when learning FP, and it shows.
The students learning Scheme at UIO show issues with their old habits causing them to favor imperative/OO approaches like loops and mutation rather then the pure FP approaches they are expected to learn. This causes them to show issues when learning and problem solving, and appears to lower their motivation for learning FP.
The issue with mutation when learning FP is not restricted to only the UIO students learning Scheme, as the lecturers note experiences both personally and with colleagues that demonstrate that even professional programmers can have their trans-

fer to FP hampered by mutation.
Successfully addressing these issues could require significant efforts, particularly for the UIO FP course which utilizes mutation extensively. Approaches to address the issue of mutation could involve postponing the introduction of mutation in FP until the students have a more solid understanding of the paradigm, or involve the conditions required for conceptual change as discussed by Posner et al[28] which suggest that shifting the students preference away from mutation could be achieved through putting the students in situations that make them dissatisfied with mutation, where a purely functional approach would present itself as a more fruitful alternative. If the teachers are successful in addressing mutation when teaching FP, this should hopefully increase students' motivation and learning retention when transferring from OO to FP.

7.5 RQ5: What do teachers think about transfer and what is their current practice?

Unsurprisingly, teachers play a rather important role in CS education. They govern the programming courses, design the curricula and hold the lectures.
This section will detail how the interviewed lecturers think of and use transfer in their teaching.

7.5.1 Lecturers' thoughts on transfer

The interviewed lecturers all held positive attitudes towards transfer. They view it as an important, useful tool for teaching and learning programming, and consider the ability to view a concept from multiple perspectives and PLs as a healthy step towards maturity as a programmer.
Transfer is seen as a natural thing which the students are likely to do instinctively, with or without the lecturers prompting, and as such teachers should take advantage of and help facilitate that transfer.
In some instances, transfer helps teachers save their efforts when teaching constructs the students are going to catch easily anyways. For instance, as mentioned by L1, there is no need to hold a separate lecture to explain the operators and operations involved in arithmetic for Scheme and Haskell, as they use all the same signs and work the same way as in every other PL. As L1 mentions, it would be crazy if they did not.
L2 also mentions how transfer can be used to demystify concepts that students find challenging, such as Haskell's monads. By showing examples of how monads resemble patterns in PLs the students are already familiar with, it can be made clear that the concept is not as alien as it appears to the students at first glance.
To be fair, teaching for transfer is not entirely without its pitfalls either. L1 mentioned an attempt to teach two similar PLs side by side, Haskell and Lifescript in order to compare and contrast them. The PLs were very similar, and as such one could almost directly paste code from Lifescript into a Haskell compile, yet still students complained that the approach was confusing.
L2 has also mentioned complaints the Haskell MOOC has received, where the many code-examples from various PLs that were used for comparison drew ire from stu-

dents who felt they have to work extra hard to understand code in PLs they do not know and did not wish to learn.
It appears some students are hesitant to put more effort into understanding PL examples that are not strictly part of the curriculum, even when they have the benefit explained to them.
However, the complaints mentioned come from students who knew neither the PL being taught, nor the PL example it was compared to. If instead they are given examples from a PL they are already familiar with, this should reduce the perceived effort in understanding transfer.

7.5.2 Lecturers' use of transfer

As the lecturers view transfer favorably, they do make attempts to help facilitate and guide students' transfer in the desired direction.
L4 has only recently taken over the UIO FP course, and as such does not yet feel comfortable making changes and putting their own spin on the course, instead focusing on catching up to the curriculum.
L1 and L2 however, have both spent years teaching Haskell at Glasgow and as such are perfectly comfortable using transfer as a tool for teaching. Their favored approach to transfer is using examples from PPLs to teach the NPL, often mentioning differences in how a given concept works in the new context compared to what students are used to previously, or showcasing code-examples side-by-side to better illustrate the syntactic and semantic differences.
They already have experience handling concepts the MPLT predicts as FCCs, such as how L2 has explicitly pointed out the similarity of C's ternary-operator to Haskell's if-expression.
The relative success L1 and L2 has enjoyed by using transfer to teach FP shows that teachers could stand to benefit by adopting a more transfer-based approach, for instance by using the MPLT to categorize concepts that the lecturers ought to take special care with when teaching, such as FCCs and ATCCs, which could help in planning what examples to use in future teaching for transfer.

7.5.3 Teaching for transfer requires knowledge of students' backgrounds

Something that became abundantly clear through this study is that teaching for transfer requires more knowledge than simply what is being taught in the given course.
In order to give examples from a different PL to help teach the course PL, the teacher has to have a certain familiarity with the PL that they are taking examples from. Beyond that, they have to be sure that the students are also familiar with that PL or else the examples are just going to fall flat, or worse, confuse the students.
In order to be a more effective teacher, they should have at least some knowledge of what experiences and backgrounds the students are bringing to the course. This can help teachers gain awareness of what the students already know, and avoid assumptions that students know what they actually do not.
If the teachers know what concepts the students are familiar with from their PPLs, this can help them pick better examples and approaches to teaching the concept in

the NPL. They can also avoid making light of what the students do not understand, such as L1's example of lecturers teaching a course on C, who erroneously assumed their students were already used to reasoning about memory management.
L1 and L2 have a very good awareness of their students' prior knowledge, but this is in large part because they have been involved in teaching the same students in previous courses, and L2 admitted to being uncertain whether they would have good awareness if this was not the case. It would be an unfair standard to expect all FP teachers to teach introductory courses simply to improve transfer for a higher-level FP course, but there are other steps that could be taken to improve lecturers' awareness of their students' backgrounds.
L1 suggested it might be advantageous for universities to set up a shared repository where lecturers could see what students had previously been taught at a given level, pointing out that it would be particularly useful to junior lecturers or lecturers taking over a new course. Given the benefits of knowing students' backgrounds, it seems rather odd that the universities do not appear to promote this awareness to their lecturers. Rather, as lamented by Pears et al 2007[29] it is left up to each lecturer to reinvent the wheel. The lecturers themselves have to discover their students' prior experiences, with little to no guidelines or facilitation on the part of their university.

7.5.4 RQ5 discussion summary

This section discussed teachers' views on transfer, which were mostly positive, although noting experiences where students they taught sometimes were confused when they felt that they were shown too many new PLs at the same time.
It also pointed out how lecturers make use of transfer when teaching, by using examples of concepts from students' PPLs, and how important it is for lecturers to be aware of students' backgrounds when teaching, particularly when teaching for transfer.
When teachers are aware of their students' prior experiences they can intentionally teach in a way that connects the new material with concepts they already understand, improving their ability to teach FP by enabling them to take better advantage of students' natural transfer.

7.6 RQ6: Are there issues of transfer beyond the MPLT?

The MPLT works well as a model for predicting PL transfer, but there are issues the model does not cover, which will be discussed in this section.
The MPLT covers transfer based on syntax and semantics, but some concepts are too abstract or large to properly account for in the model.

7.6.1 Paradigms

One could reasonably define the syntax of certain abstract concepts such as recursion or higher-order functions, due to their more concrete expressions, but how do you define the syntax of FP as a paradigm?
There is simply too much that goes into a paradigm to reasonably limit it to some

keyword that can be predicted based on its syntax and semantics.
The transfer of paradigms is important however, as observed with the issues revolving mutation. The students are used to thinking in OO, and the interviews show that they bring this way of thinking into FP.
L4 especially was concerned with students who were used to one paradigm fixated on making their old approaches work in the new paradigm, ignoring the new opportunities offered by the paradigm has to offer.

> "You learn a new language and you still think in the old one so that means you somehow go “Oh no, I need to use patterns, because that’s super cool” and then you try to use inheritance in Lisp, or in another language, not realizing that the other patterns that you could use, which would have addressed the problem in a different way more suitable to the new language" [L4]

This habit of getting stuck in old ways of thinking are important to address, but the MPLT would struggle to predict exactly how this transfer would happen when it comes to the paradigms themselves.

7.6.2 Abstract concepts

Similar to shifting paradigms, there are some concepts that cannot be properly captured by the MPLT given their highly abstract nature.
While the concept of a higher-order function is rather abstract, it is simple enough to throw out examples like *map* and *fold* with more clearly defined syntax and semantics for the students to transfer.
However, some concepts are not so clearly exemplified as that. Mutation is an example of this type of concept, where the concept as a whole cannot be summed up in just syntax and semantics, but as a whole mindset for thinking about programming. While the MPLT could reasonably predict how students would transfer certain concepts that make mutation possible, such as predicting Scheme’s *set!* as an ATCC due to its dissimilar syntax to Java and Python, the way students would wholeheartedly throw themselves into using mutation to solve as many problems as they possibly could, would not be predictable by the MPLT.

7.6.3 RQ6 discussion summary

While the MPLT is a validated and useful model for predicting PL transfer, it is important to keep in mind that it does not cover every facet of transfer.
For certain large, abstract concepts it is difficult to accurately use the model to predict how students will transfer, such as mutation or FP as a paradigm.
The transfer of these concepts is also worthy of exploration to better understand and plan for how students will learn FP.

7.7 RQ7: How could exploring transfer aid in teaching FP?

This section looks at the different teaching methods associated with the three transfer categories of the MPLT and how they aid in teaching for transfer, as well as how awareness of students' transfer will aid lecturers in teaching FP.

7.7.1 Teaching TCCs, FCCs and ATCCs

The different categories of transfer predicted by the MPLT are not simply a handy way of sorting different concepts, they are also predictions of the neccessary efforts needed by students to learn these concepts in the NPL, and by extension, predictions of the efforts needed to teach these concepts.
When a student comes across a TCC, they will assume the concept to be similar to a concept they knew from a PPL, because of the shared syntax. In this case, the assumption will be correct, as the underlying semantics of the concept also closely match the concept they were used to. Teaching a TCC to students is less of an effort to explain the concept, and more of getting out of the way of the students and letting them make the connections on their own. The transfer of a TCC is predicted to be so natural that the students will instinctively transfer them even without aid.
FCCs are a lot trickier however. The problem with FCCs is that, same as TCCs, the students will assume the concept is similar to a concept they knew from a PPL, because of shared syntax. The issue here is that, while the syntactic structure is near the same, the underlying semantics can be vastly different, leaving the students stuck with misconceptions that can be difficult to self-correct. Some misconceptions students will eventually address on their own by trial and error, but some can be far more sinister. It can for instance be easy to assume that Haskell's list-range functions the same way as list-ranges in Java and Python, but creating a list-range from 1 to 5 in Haskell outputs [*1, 2, 3, 4, 5*] while in Python and Java the same would output [*1, 2, 3, 4*]. It is important that the lecturer correctly identifies and addresses FCCs as quickly as possible, to avoid students falling into these syntactic traps and entrenching their misconceptions. Tshukudu et al [16] proposes addressing FCCs by explicitly comparing the FCC to the PPL to make students aware that they do not work the same way.
ATCCs are not as dangerous to students' learning as FCCs, but they hold great potential for transfer learning. Because the students do not recognize the syntax of the NPL concept, they assume that the concept is novel and make little to no attempt on their own to connect their prior experience to aid in learning the ATCC, despite the semantics being similar to a concept they are already familiar with. Teaching an ATCC requires more work than a TCC, because the students are not likely to make the connection themselves, but the effort does not have to go far beyond simply pointing out the similarity of the ATCC to the similar concept in a PPL. Once the connection has been made, the students will be able to transfer the concept on their own.

7.7.2 Awareness of what students transfer will help lecturers teach FP

Before taking the FP courses at UIO and Glasgow, students will have taken multiple courses beforehand. In fact, for all non-introductory courses students will have prior experiences, and some even have experience programming from high-school or earlier.
The status quo of the universities currently seems to be letting each lecturer muddle

through individually, with little to no thought on providing them with a helpful overview of what previous languages and experiences the students bring with them. Some lecturers do a great job of looking into their students' backgrounds and benefit immensely, whereas others do not take the effort, are not aware that they should, or even know how to do it if they wanted to.
Knowing what students know before coming to class can aid a teacher immensely in planning their curriculum accordingly. They can account for FCCs that students are likely to misunderstand, and help the students make connections to ATCCs they would not make themselves.
Conversely, not being aware of transfer can harm students' enjoyment of a given course, as the teacher might assume knowledge the students have, as L2 mentioned regarding the C-lecturers who assumed students reasoned about memory, or L4 who used examples from a compiler course they held without being aware that none of the students had taken the course.
Had the teacher who initially created the FP course at UIO been aware of the negative impact mutation has on learning FP, they might have structured the course differently to mitigate the potential harm on students' learning.

7.7.3 RQ7 discussion summary

Being aware of transfer and its consequences would help teaching FP, as lecturers can better plan their curricula to account for the way concepts transfer, and take steps in advance to avoid or address problematic transfer such as FCCs or mutation. By improving the teachers ability to plan and teach for transfer, they will have an easier time teaching FP to their students, and can be more confident that their students will actually learn from what they are teaching.

Chapter 8

Future research

This chapter focuses on areas of research that would be highly relevant to explore in more detail, but unfortunately could not be fully covered during the course of this project.

8.1 Compare transfer to different FP languages

This project looked at students with OO backgrounds transferring to two different FP languages, aimed at exploring transfer and the efficacy of the MPLT. However, it could also be interesting to perform a study specifically focused on comparing how students with an OO background transfers to an FP language f. ex Haskell in comparison with how students with a similar background transfer to another FP language. By comparing how students with similar OO backgrounds transfer to different FP languages, it becomes possible to observe what impact the specifics of the PL has on transfer, and what impact the FP paradigm as a whole has on transfer.

8.2 Further exploring the influence of mutation on learning FP

While not a primary goal of this project, the conducted studies revealed highly interesting results about the negative influence mutation has on students learning FP. The next step in uncovering the full extent of this influence would be conducting a study focused on when, how and to what extent mutation interferes with students' learning and motivation for FP.

Chapter 9

Conclusion

This thesis project used interviews and quizzes to explore PL transfer as well as the validity of the MPLT as a model for predicting student transfer from OO to FP.
The interviews and quizzes demonstrated that students do in fact tend to fixate on using syntax matching to transfer a given concept's semantics. Given this syntactic focus, the three categories of transfer used by the MPLT appear to accurately predict intermediate students' transfer from OO Python and Java to FP Haskell and Scheme, although some concepts have proved challenging to predict and/or test.
The interviews with students, teacher's assistants and lecturers have shed further light on how intermediate students transfer, and how teachers account for and facilitate that transfer. The study has shown that intermediate students with prior transfer experience has a somewhat more flexible approach to transfer than novice students, seeing an improvement to their ability to transfer new concepts that does not share similar syntax with concepts from a previously learnt programming language when compared to the novices. The intermediate students also demonstrate more CS knowledge in general, enabling them to make connections between concepts the novices are wholly unaware of.
An important finding of this project have been the great, negative impact mutation can have on students with an OO background transferring to FP, as students have proved likely to prefer relying on their familiar approach of using mutation to solve problems to the detriment of alternative, purely functional solutions.
The teachers and teaching assistants interviewed for the study have shown positive attitudes towards using transfer to teach FP, but the status quo of the universities involved has so far been to leave the awareness of students' backgrounds and effective use of transfer, or lack thereof, entirely in the hands of the individual teacher.
Backed up by the project's findings, it is the concluding remarks of this thesis that exploring transfer can improve the process of teaching and learning FP, and the MPLT has proven a valid, useful tool to aid this exploration.

Bibliography

[1] Ethel Tshsukudu and Quintin Cutts. Understanding conceptual transfer for students learning new programming languages. *ICER*, '20:227–337, 2020.

[2] Ethel Tshukudu and Quintin Cutts. *Understanding conceptual transfer for students learning new programming languages.* PhD thesis, University of Glasgow, 2020.

[3] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3:49–65, 1993.

[4] Ville Tirronen, Samuel Uusi Mäkelä, and Ville Isomöttönen. Understanding beginner's mistakes with haskell. *Journal of Functional programming*, 25:1–30, 2015.

[5] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *ITiCSE*, '05:14–18, 2005.

[6] Nischal Shreshta, Colton Botta, Titus Barik, and Chris Parnin. Here we go again: Why is it difficult for developers to learn another programming language. *ICSE*, '20:1–7, 2020.

[7] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. *OOPSLA*, '13:1–18, 2013.

[8] Danny Kopec, Gavriel Yarmish, and Patrick Cheung. A description and study of intermediate student programmer errors. *ACM SIGCSE Bulletin*, 39(2):146–156, 2007.

[9] Amnon Shabo, Mark Guzdial, and John Stasko. Computer science apprenticeship: Creating support for intermediate computer science students. *AACE*, '96:308–315, 1996.

[10] Adrienne Decker and David Simkins. Uncovering difficulties in learning for the intermediate programmer. *IEEE*, pages 1–8, 2016.

[11] Matt Bower and Annabelle McIver. Continual and explicit comparison to promote proactive facilitation during second computer language learning. *ITiCSE*, '11:218–222, 2011.

[12] Igor Moreno Santos, Matthias Hauswirth, and Nathaniel Nystrom. Experience in bridging from functional to object-oriented programming. *SPLASH-E*, '19:36–40, 2019.

[13] Ethel Tshukudu and Quintin Cutts. Semantic transfer in programming languages: Exploratory study of relative novices. *ITiCSE*, '20:307–313, 2020.

[14] Ethel Tshsukudu and Siri Annethe Moe Jensen. The role of explicit instructions on students learning their second programming language. *UKICER*, '20:10–16, 2020.

[15] Ethel Tshsukudu, Quintin Cutts, Olivier Goletti, Alaaeddin Swidan, and Felienne Hermans. Teachers' views and experiences on teaching second and subsequent programming languages. *ICER*, '21:294–305, 2021.

[16] Ethel Tshsukudu, Quintin Cutts, and Mary Ellen Foster. Evaluating a pedagogy for improving conceptual transfer and understanding in a second programming language learning context. *Koli Calling*, '21:1–10, 2021.

[17] Nan Jiang. Lexical representation and development in a second language. *Applied linguistics*, 21(1):47–77, 2000.

[18] L. V. Tulchak and M. O. Marchuk. *History of Python.* PhD thesis, Vinnytsia National Technical University, 2016.

[19] Sloan Kelly. *Python, PyGame and Raspberry Pi Game Development.* Springer, 2016.

[20] James Gosling, David Colin Holmes, and Ken Arnold. *The Java programming language.* Addison-Wesley, 2005.

[21] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[22] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An interpreteter for extended lambda calculus. *AI memo no. 349*, pages 1–42, 1975.

[23] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. *ACM SIGPLAN*, pages 12–1, 2007.

[24] Virginia Braun and Victoria Clarke. *Thematic analysis*, volume 2. American Psychological Association, 2012.

[25] Jean Scholtz and Susan Wiedenbeck. An analysis of novice programmers learning a second language. *PPIG*, 1:9, 1992.

[26] Jean Scholtz and Susan Wiedenbeck. Learning a new programming language: a model of the planning process. *IEEE*, 2:3–12, 1991.

[27] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs.* The MIT Press, 1996.

[28] George J. Posner, Kenneth A. Strike, Peter W. Hewson, and William A. Gertzog. Toward a theory of conceptual change. *Science education*, 66(2):211–227, 1982.

[29] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *ITiCSE*, 39(4), 2007.

Appendix A

Guess quizzes

Attached are the two guess quizzes administered to students, with the Haskell guess quiz administered to the student at the University of Glasgow, and the Scheme guess quiz administered to intermediate and novice students at the University of Oslo.

A.1 Haskell guess quiz

The below guess quiz was created and administered using Moodle by the two lecturers responsible for the FP course at the University of Glasgow.

In the following exercise, we are asking you to explain lines of Haskell in the full knowledge that you haven't started to learn Haskell yet. Give the best answers you can to the following questions, despite this. We will be analyzing your responses and using them to tailor our teaching - so the more attention you pay to answering, the better we will be able to support your learning.

For each of the questions A to E, which ask you to explain some line(s) of Haskell code, please consider your answer in three ways:
i. What the line(s) of code will do when executed - what will be the outcome?
ii. What you believe the constituent parts of the line(s) are - what computing names would you give them? e.g. function, name, parameter, argument, value - and how do they work/operate?
iii. How you came to this explanation - did the line(s) remind you of code in other languages? If so, say which languages?

A. Explain as much as you can about lines 13-16. (remember i, ii, and iii above)
B. Explain as much as you can about line 17 - explain what each part of the line is, and what you think will happen when it is executed. (remember i, ii, and iii above)
C. Explain as much as you can about line 9. (remember i, ii, iii)
D. Without going into too much detail, explain lines 11-24 (ha ha - i, ii, iii again!)
E. Explain as much as you can about the lines 6-8 (i, ii, and iii)

```
1 module Main where
2 combineSentences1 [s1, s2] = unwords $ map longestWord (zip (words s1) (words
s2))
```

```
combineSentences1 _ = error "Can only combine two sentences"
combineSentences2 [s1, s2] = unwords (combineSentences2Rec (zip (words s1) (words s2)))
combineSentences2 _ = error "Can only combine two sentences"
combineSentences2Rec [] = []
combineSentences2Rec [s12] = [shortestWord s12]
combineSentences2Rec (s12:s12s) = [shortestWord s12] ++ combineSentences2Rec s12s
longestWord (w1,w2) = if length w1 > length w2 then w1 else w2
shortestWord (w1,w2) = if length w1 < length w2 then w1 else w2

main = do
let
sentences = [
"The quick brown fox jumped over the low wall.",
"The startled red deer jumps over the high hedge."
]
putStrLn "Sentence 1:"
putStrLn (head sentences)
putStrLn "\nSentence 2"
putStrLn $ sentences !! 1
putStrLn "\ncombineSentences1:"
putStrLn $ combineSentences1 sentences
putStrLn Ex"\ncombineSentences2:"
putStrLn $ combineSentences2 sentences
```

A.2 Scheme guess quiz

The below guess quiz was created by the author of this thesis and administered using Nettskjema to the students studying FP, as well as novice students, at the University of Oslo.

Nettskjema

Guess quiz IN2040

Did you have programming experience before you started at UIO?

Yes

No

Pick the alternative below that best approximates your level of prior experience

This element is only shown when the option 'Yes' is selected in the question 'Did you have programming experience before you started at UIO?'

Beginner - Wrote Hello-world at some point

Hobby - Coded a few small projects

Enthusiast - Resident tech-wizard

Professional - I&#39;ve worked as a programmer for 1&#43; years

What programming languages do you use?

Note down any language you could use when doing a work-, school- or hobby-related project
(Please rank the languages in order of familiarity - 1. Java, 2. C, 3. Python, etc.)

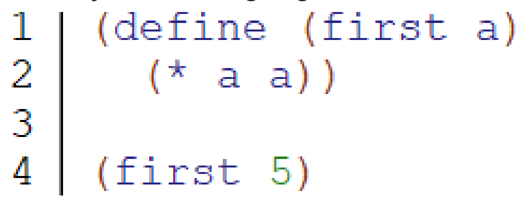
In the following exercise, we are asking you to explain lines of Scheme in the full knowledge that you haven't started to learn Scheme yet. Give the best answers you can to the following questions, despite this. We will be analyzing your responses and using them to tailor our teaching - so the more attention you pay to answering, the better we will be able to support your learning.

There are 4 questions. For each of the questions 1 to 4, which ask you to explain some line(s) of Scheme code, please consider your answer in three ways:

i. What the line(s) of code will do when executed - what will be the outcome/result?

ii. What you believe the constituent parts of the line(s) are - what computing names would you give them? e.g. function, name, parameter, argument, value - and how do they work/operate?

iii. How you came to this explanation - did the line(s) remind you of code in other languages? If so, say which languages?



```
1 | (define (first a)
2 |   (* a a))
3 |
4 | (first 5)
```

1. Try to explain as much as you can about line 1-4. (remember i, ii, and iii above)

In the following exercise, we are asking you to explain lines of Scheme in the full knowledge that you haven't started to learn Scheme yet. Give the best answers you can to the following questions, despite this. We will be analyzing your responses and using them to tailor our teaching - so the more attention you pay to answering, the better we will be able to support your learning.

There are 4 questions. For each of the questions 1 to 4, which ask you to explain some line(s) of Scheme code, please consider your answer in three ways:

i. What the line(s) of code will do when executed - what will be the outcome/result?

ii. What you believe the constituent parts of the line(s) are - what computing names would you give them? e.g. function, name, parameter, argument, value - and how do they work/operate?

iii. How you came to this explanation - did the line(s) remind you of code in other languages? If so, say which languages?

```
6 | (define numbers (list 1 2 3 4))
```

2. Try to explain as much as you can about line 6. (remember i, ii, and iii above)

In the following exercise, we are asking you to explain lines of Scheme in the full knowledge that you haven't started to learn Scheme yet. Give the best answers you can to the following questions, despite this. We will be analyzing your responses and using them to tailor our teaching - so the more attention you pay to answering, the better we will be able to support your learning.

There are 4 questions. For each of the questions 1 to 4, which ask you to explain some line(s) of Scheme code, please consider your answer in three ways:

i. What the line(s) of code will do when executed - what will be the outcome/result?

ii. What you believe the constituent parts of the line(s) are - what computing names would you give them? e.g. function, name, parameter, argument, value - and how do they work/operate?

iii. How you came to this explanation - did the line(s) remind you of code in other languages? If so, say which languages?

```
 6 | (define numbers (list 1 2 3 4))
 7 |
 8 | (define (second lst)
 9 |     (if (null? lst)
10 |         '()
11 |         (cons (+ (car lst) 1) (second (cdr lst)))))
12 |
13 | (second numbers)
```

3. Try to explain as much as you can about line 8-13. (remember i, ii, and iii above)

a) What's the purpose of line 8?

b) How are lines 9-11 executed, and what are the constituent parts?

c) Try to explain line 13

Other observations/notes?

In the following exercise, we are asking you to explain lines of Scheme in the full knowledge that you haven't started to learn Scheme yet. Give the best answers you can to the following questions, despite this. We will be analyzing your responses and using them to tailor our teaching - so the more attention you pay to answering, the better we will be able to support your learning.

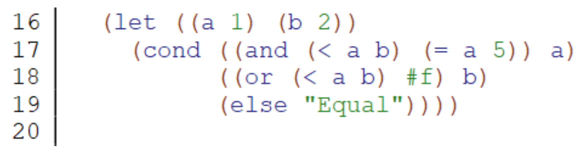
There are 4 questions. For each of the questions 1 to 4, which ask you to explain some line(s) of Scheme code, please consider your answer in three ways:

Nettskjema

i. What the line(s) of code will do when executed - what will be the outcome/result?

ii. What you believe the constituent parts of the line(s) are - what computing names would you give them? e.g. function, name, parameter, argument, value - and how do they work/operate?

iii. How you came to this explanation - did the line(s) remind you of code in other languages? If so, say which languages?



```
16    (let ((a 1) (b 2))
17      (cond ((and (< a b) (= a 5)) a)
18            ((or (< a b) #f) b)
19            (else "Equal"))))
20
```

4. Try to explain as much as you can about line 16-20. (remember i, ii, and iii above)

a) What's the purpose of line 16?

b) How are lines 17-19 executed, and what are the constituent parts?

Other observations/notes?

Appendix B

Interview guides

Interview guides were written for interviews with students, TAs and lecturers. There were two interview guides created for lecturers, with one for the lecturers at UIO and one for the lecturers in Glasgow.

B.1 Interview guide students

Below is the interview guide written for the students, including questions about necessary background information and the program comprehension and coding tasks they were asked to perform. Although the interviews were held in Norwegian, the interview guide was written in English to benefit the readability of non-Norwegian readers.

Name:

Email:

Background

How many years have you studied programming?

What programming languages are you proficient with?

- How proficient would you say you are with each language?

Code

Code example 1:

What do you see here? Does this remind you of any concept in another programming language? Which? What do you think the output will be?

```
(define (something n)
  (let ((a 10))
    (+ a n)))

(something 5)
(something -10)
(something 10)
```

Response:

Code example 2:

What do you see here? Does this remind you of any concept in another programming language? Which? What do you think the output will be?

```
(cond ((and (< 1 2) #f) 1)
      ((or (> 1 2) 5) 2)
```

```
    (else "Hello"))
```

Response:

Code example 3:

What do you see here? Does this remind you of any concept in another programming language? Which? What do you think the output will be?

```
(define (some-func l)
  (if (null? l)
      '()
      (cons (+ (car l) 1) (some-func (cdr l)))))

(some-func '(1 2 3 4 5))
```

Response:

Code exercise 1

Write a function in Scheme that calculates the length of a list

- How would you do this in Java/Python?

What did you think when you first saw function definitions and function calls in Scheme? Did you recognize them, if so, what did they remind you of?

When writing recursive functions, do you prefer working with iterative or recursive processes? Why?

Course

How do you like the course?

What was it like to see functional programming for the first time?

How do you think Functional programming works now?

Have you had to change the way you think of programming when programming in Scheme?

How did you start learning Scheme?

What strategies did you use to learn Scheme?

Does Scheme remind you of any other language(s) you know? If so, which?

What has been the hardest concept(s) to learn?

- Has there been any conflicts with what you already knew from another language?

Have you had any difficulties?

What concept(s) has been easiest to learn?

- Did you have any prior experience with this concept?

Have you had any moments where you've thought: This reminds me a lot of that concept in another language? What concept and language was that?

- How similar were the concepts?
- Did you find the experience helpful or detrimental?

B.2 Interview guide Teaching Assistants

The interviews with TAs involved the same program comprehension and coding tasks as those of the students. Similar to the interviews with students the interview guide was written in English to benefit the readability of non-Norwegian readers, while the actual interviews were held in Norwegian.

Interview guide teachers assistants

Background

What is your name?

What email can I use to contact you?

How long have you been a teacher’s assistant in this course?

When did you take this course yourself?

Have you since had any other experience that used what you learned from this course? Other courses? Projects?

How many years have you been programming?

What other programming languages are you experienced with?

Programming tasks

Code example 1:

What do you see here? Does this remind you of any concept in another programming language? Which? What do you think the output will be?

```
(define (something n)
  (let ((a 10))
    (+ a n)))

(something 5)
(something -10)
(something 10)
```

Response:

Code example 2:

What do you see here? Does this remind you of any concept in another programming language? Which? What do you think the output will be?

```
(cond ((and (< 1 2) #f) 1)
       ((or (> 1 2) 5) 2)
      (else "Hello"))
```

Response:

Code example 3:

What do you see here? Does this remind you of any concept in another programming language? Which? What do you think the output will be?

```
(define (some-func l)
  (if (null? l)
      '()
      (cons (+ (car l) 1) (some-func (cdr l)))))

(some-func '(1 2 3 4 5))
```

Response:

Code exercise 1

Write a function in Scheme that calculates the length of a list

- How would you do this in Java/Python?

Programming language transfer

What does the word “Transfer” mean to you?

What do you think the word means in the context of learning a Programming Language?

Are you aware of any concepts in Scheme that made you think “this is similar to that other concept in Python/Java”

Can you think of some concepts where knowing the concept in Python/Java could lead to misunderstandings when learning the concept in Scheme? f.ex if

Are you conscious of transfer when teaching?

- If so, how do you take advantage of transfer?
- If not, how do you think you could take advantage of transfer?

Do you notice any programming language transfer in the students from your group when teaching functional programming?

- If so, can you give examples of concepts that transfer?

Have you noticed any transfer when correcting assignments?

- Have you seen any students fail a task because they conflate a concept with another from a different language? (f.eks if)

Have you noticed your students misunderstanding a concept due to conflicting prior experience?

B.3 Interview guide IFI lecturers

The interviews with lecturers had no coding tasks, only theoretical questions. One of the interviews were conducted in Norwegian, the other was conducted in English.

Interview guide lecturers IFI

Background

What is your name?

What email can I use to contact you?

How long have you taught functional programming?

How long have you been programming?

What other programming languages are you experienced with?

What language(s) are you most familiar with?

Programming language transfer

What do you think of transfer in the context of learning programming languages?

Are you conscious of transfer when creating the curriculum and/or while teaching?

If so, how do you take advantage of transfer? If not, how would you plan to?

How aware are you of what your students have learned before taking your course?

What do you do with that information?

Can you give examples of concepts that transfer?

Do you think transfer can be negative? Why/why not?

Do you know why IN2040 has such a focus on imperative/object-oriented concepts?

What do you think of that focus?

Are you aware of the MPLT, a model for predicting transfer?

What are your thoughts on it?

When creating or solving tasks with functional programming, do other programming languages influence your thinking?

How do you think the mandatory assignments will affect students' transfer?

Anything else?

B.4 Interview guide Glasgow lecturers

Due to the different context for the lecturers from different universities, some questions were changed between the guides. Both interviews were conducted in English.

Interview guide lecturers Glasgow

Background

What is your name?

What email can I use to contact you?

How long have you taught functional programming?

How long have you been programming?

What other programming languages are you experienced with?

What language(s) are you most familiar with?

Programming language transfer

What do you think of transfer in the context of learning programming languages?

Are you conscious of transfer when creating the curriculum and/or while teaching?

If so, how do you take advantage of transfer? If not, how would you plan to?

How aware are you of what your students have learned before taking your course?

What do you do with that information?

Can you give examples of concepts that transfer to Haskell?

Do you think transfer can be negative? Why/why not?

What do you think of that focus?

Are you aware of the MPLT, a model for predicting transfer?

What are your thoughts on it?

When creating or solving tasks with functional programming, do other programming languages influence your thinking?

How do you think the mandatory assignments will affect students’ transfer?

Anything else?

Appendix C

Consent form

Below is the consent form all interviewees were asked to sign before the start of the interviews, asking for consent to record the voices of those interviewed for later transcription.

Are you interested in taking part in the research project

”Investigating Intermediate Student Transfer to Functional Programming Languages”?

This is an inquiry about participation in a research project where the main purpose is to Investigate how Intermediate Students transfer conceptual knowledge from Object-oriented programming to Functional programming. In this letter we will give you information about the purpose of the project and what your participation will involve.

Purpose of the project

This master’s project aims to gather data on how intermediate students learn a new programming language by adapting experience working with other programming languages. In

this context, bachelor students at the University of Oslo and the University of Glasgow will be studied to see how their experience working mostly with Object-Oriented programming languages impacts their ability to learn Functional programming languages. This project also aims to examine the MPLT's (Model for Programming Language Transfer) effectiveness in predicting the impact prior Object-Oriented experience has on students learning Functional programming.
During the course of the project, students will be asked to participate in a small number of guess quizzes, where no personal data is sought, and participation is voluntary. A small number of students will also be asked to participate in a series of one-on-one sessions to further examine how their learning experience is impacted by transfer. In addition, lecturers and teacher's assistants will be asked to participate in interviews detailing their experiences working with programming language transfer.

Who is responsible for the research project?

The University of Oslo is the institution responsible for the project, in cooperation with the University of Glasgow

Why are you being asked to participate?

You have been asked to participate in an interview because you teach IN2040: Funksjonell Programmering, either as a lecturer or teacher's assistant

What does participation involve for you?

- *« If you choose to take part in the project, this will involve that you participate in an interview lasting roughly 45 minutes. Your responses will be recorded electronically as a sound recording.*

Participation is voluntary

Participation in the project is voluntary. If you chose to participate, you can withdraw your consent at any time without giving a reason. All information about you will then be made anonymous. There will be no negative consequences for you if you choose not to participate or later decide to withdraw.

Your personal privacy – how we will store and use your personal data

We will only use your personal data for the purpose(s) specified in this information letter. We will process your personal data confidentially and in accordance with data protection legislation (the General Data Protection Regulation and Personal Data Act).

- *The only people who will have access to your data are the project group, eg. student and supervisors*
- *Your personal data will be stored on UIO's servers*

What will happen to your personal data at the end of the research project?

The project is scheduled to end *30.12.2023. By this date, all sound recordings will be deleted.*

Your rights

So long as you can be identified in the collected data, you have the right to:

- access the personal data that is being processed about you
- request that your personal data is deleted
- request that incorrect personal data about you is corrected/rectified
- receive a copy of your personal data (data portability), and
- send a complaint to the Data Protection Officer or The Norwegian Data Protection Authority regarding the processing of your personal data

What gives us the right to process your personal data?

We will process your personal data based on your consent.

Based on an agreement with *the University of Oslo*, Data Protection Services has assessed that the processing of personal data in this project is in accordance with data protection legislation.

Where can I find out more?

If you have questions about the project, or want to exercise your rights, contact:

- *University of Oslo* via Professor *Quintin Cutts, by email: (quintin.cutts@glasgow.ac.uk) or by telephone: +44* 01413305619
- *Or via Jørgen Spilling, by email: (jorgensp@uio.no) or by telephone (+47 94 48 85 08)*
- Data Protection Services, by email: (personverntjenester@sikt.no) or by telephone: +47 53 21 15 00.

Yours sincerely,

Quintin Cutts Jørgen Spilling

(Researcher/supervisor)

Consent form

I have received and understood information about the projectInvestigating Intermediate Student Transfer to Functional Programming Languages and have been given the opportunity to ask questions. I give consent:

- to participate in *an interview*

I give consent for my personal data to be processed until the end date of the project, approx. *30.12.2023*

(Signed by participant, date)