JFP **25**, e11, 30 pages, 2015. © Cambridge University Press 2015
doi:10.1017/S0956796815000179

Understanding beginners' mistakes with Haskell

VILLE TIRRONEN, SAMUEL UUSI-MÄKELÄ and
VILLE ISOMÖTTÖNEN

Department of Mathematical Information Technology, University of Jyväskylä,
P.O. Box 35 (Agora), 40014, Jyväskylä, Finland
(*e-mail:* ville.tirronen@jyu.fi, samuel.h.uusi-makela@student.jyu.fi,
ville.isomottonen@jyu.fi)

Abstract

This article presents an overview of student difficulties in an introductory functional programming (FP) course taught in Haskell. The motivation for this study stems from our belief that many student difficulties can be alleviated by understanding the underlying causes of errors and by modifying the educational approach and, possibly, the teaching language accordingly. We analyze students' exercise submissions and categorize student errors according to compiler error messages and then manually according to the observed underlying cause. Our study complements earlier studies on the topic by applying computer and manual analysis while focusing on providing descriptive statistics of difficulties specific to FP languages. We conclude that the majority of student errors, regardless of cause, are reported by three different compiler error messages that are not well understood by students. In addition, syntactic features, such as precedence, the syntax of function application, and deeply nested statements, cause difficulties throughout the course.

1 Introduction

Beginners' difficulties in programming have been extensively reported in the literature. Reports on imperative languages point to difficulties with abstract concepts and program construction (Lahtinen *et al.*, 2005), problem solving, testing, debugging, and documentation (Ulloa, 1980), as well as problem analysis (Someren, 1990) and the interpretation of error messages (Marceau *et al.*, 2011b). More specifically, one problem seems to be that students attempt to apply naive or natural language-like solutions that are not appropriate for use with programming languages (Someren, 1990; Pane *et al.*, 2001). Some studies suggest that difficulties in programming courses may be rooted in personal and social issues (Tinto, 1997; Kinnunen & Malmi, 2006), while, at the other end of the research spectrum, beginners' difficulties have been addressed by identifying non-viable mental models that cause misconceptions and difficulties (Ma *et al.*, 2011).

FP languages are a strong choice for teaching introductory programming (Felleisen *et al.*, 2001; Findler *et al.*, 2002; Felleisen *et al.*, 2004), and they have been extensively studied in the context of the Lisp-family of languages (Felleisen *et al.*, 2004; Bieniusa *et al.*, 2008; Morazán, 2011; Morazán, 2012). In the present article, we are concerned with difficulties in the learning FP, specifically in the context of statically typed

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

ML-like languages. The literature provides several examples of the use of such functional languages as the main teaching tool (Joosten *et al*., 1993; Keravnou, 1995; Thompson & Hill, 1995; Blanco *et al*., 2009), while this flavor of FP has been found to be difficult for students (Chakravarty & Keller, 2004). Studies indicate that students struggle with modern type systems (Clack & Myers, 1995), recursion (Segal, 1994), and the ML-style syntax (Joosten *et al*., 1993). One known challenge is caused by the transition from imperative languages to a functional language (Joosten *et al*., 1993; Clack & Myers, 1995).

The principal motivation for our educational study comes from the indication that learners' errors in functional languages can be different from those in mainstream languages due to different programming constructs. Many FP languages, especially those of the ML family, have not been deeply studied from the viewpoint of learning them for the first time. Instead, arguments for, and especially against, the use of FP languages are often personal opinions and imagined difficulties (Vujoševic-Janicic, Milena and Tošic, Dušan, 2008) instead of data. However, since teachers' opinions on learners' difficulties are not always completely in line with reality, even in mainstream languages (Brown & Altadmri, 2014) it seems important to study the actual difficulties encountered by learners. Understanding the actual mistakes allows for their mitigation and informs us how to teach the topic.

In this paper, our research interest is to develop an overview of the mistakes made by students who are learning Haskell as their first FP language and to document how these mistakes are presented as error messages. We use student exercise answers as the research data, and by inspecting compiler error messages, we divide the answers into specific categories that characterize different kinds of mistakes. The error categories are then subjected to a detailed manual analysis and classified into subcategories that further characterize student mistakes. Finally, in the discussion section, we draw conclusions for educators regarding how to improve teaching FP and briefly discuss the implications of our findings for programming language designers who intend to support novices. The results of this study can be compared with similar studies on imperative languages (e.g., Jadud, 2005) and functional languages (e.g., Heeren *et al*., 2003; Hage & Keeken, 2006).

2 Related work

This article's field of study concerns the programming errors that beginner programmers tend to make and the reasons why such errors occur. This field has been extensively studied for more than 30 years, both in the context of expert and beginner programmers. An early example is the pioneering work by the Cognition and Programming Group at Yale University, led by Soloway *et al.* (1982), Soloway & Ehrlich (1984), and Someren (1990). These scholars formulate programming activities as a *goal/plan* process, where programmers attempt to achieve their goals through mental *plans* (tacit knowledge) that manifest as program code. In this theory, programming errors arise from the breakdown of the plans, such as applying inappropriate plans drawn from "preprogramming knowledge" (Bonar & Soloway, 1985). The still-timely findings of the Yale group posit that *misconceptions*

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

about language constructs are not the main cause for programming errors (Spohrer & Soloway, 1986). Instead, Spohrer and Soloway suggest that other errors, such as boundary problems (e.g., off-by-one error), problems in composing plans (e.g., misplaced program code, statements outside conditionals, etc.), misusing logical operators (confusing logical `and` with logical `or`), and interpretation problems (misinterpreting how the computations are performed), are the main causes of errors.

Regarding the present study, we note that many of the technical difficulties documented by the Yale group are highly dependent on the studied programming paradigm. For example, to have an off-by-one error, the program must include a numerically indexed container. In a similar vein, having a more complex model of the language will necessarily result in more difficulties in interpreting the programs.

Another model for programming errors from that period was put forth by Perkins & Martin (1986), who observed that beginners often suffer from "fragile knowledge," that is, knowledge that is either incomplete or missing entirely, inert so that student cannot recall it when required, misplaced in that it gets applied in the wrong situations, or conglomerated so that the student attempts to use disparate elements together. To relate the concepts of fragile knowledge to the present study, we suggest that various types of fragile knowledge can be more prevalent depending on the programming language used; for example, highly abstract concepts are harder to apply in specific situations, leading to increased inert knowledge. Similarly, if the syntax of the language is homogeneous, it is reasonable to assume that the constructs of the language would have a higher tendency to conglomerate (cf. (Felleisen *et al.*, 2004)).

Another contemporary explanation for the prevalence of beginners' programming errors was given by Pea (1986). In an attempt to identify language-independent beginner programmer errors, Pea documented three classes of conceptual mistakes: parallelism, intentionality, and egocentrism mistakes. Pea suggested that these classes are different facets of the so-called superbug–the idea that the computer is capable of intelligently interpreting the programmer's intent. Like his contemporaries introduced above, Pea studied the languages of the imperative paradigm. Although his main theme is certainly relevant for other paradigms, his specifics must be adjusted. For example, Pea noted that the "parallelism" misconception, evident in the novice programmer who thinks that executed statements somehow remain active for the duration of the program, is a correct mental model in some paradigms, such as logic programming.

The effects of different programming paradigms on programming errors are gradually being accounted for or even specifically studied. For example, concurrent programming errors have been categorized by Farchi *et al.* (2003), and Lu *et al.* (2008). The FP paradigm is similarly receiving increased attention. For example, Chambers *et al.* (2012) conducted an observational study of the errors made by novice functional programmers during a programming project. The authors presented several specific error categories, the most prevalent of which were "signature" errors, which we understand to refer mainly to type errors. Signature errors comprised 46% of all errors. The next most prevalent category consisted of run-time

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

errors and wrong program output, which amounted to 21% of the errors. Finally, 16% were parse errors, and 14% were due to referring to undeclared functions.

Heeren *et al.* (2003) and Hage & Keeken (2006) collected a large set of student programs over several iterations of their first-year FP course, and documented statistics such as error frequencies. The authors note that, at 30%, type errors were the most common compile time failure, followed in frequency by various syntactic errors, which were found in 20% of the student attempts. The remaining static errors, which occurred in 10% of the compilation attempts, were mostly made by referring to undefined variables.

There are many different ways to study beginner programmer difficulties. The various ways include studying videotaped "think-aloud" sessions (Soloway *et al.*, 1982), analyzing students' process journals (Lewandowski, 2003), and studying a record of the compilation errors generated by beginner programs. In this article, we apply the latter method of study. A good example of such a study is that by Jadud (2005), who documented the kinds of compilation errors that arise when students learn the Java language. Jadud's results indicate that syntactical errors, such as missing semicolons or misspellings of variable names, are the most common type of beginners' Java errors. This study was repeated by Fenwick *et al.* (2009), who obtained similar results. Fenwick *et al.* also presented higher-level patterns of student behavior, confirming, for example, the benefits of starting projects early and working incrementally.

Denny *et al.* (2012) presented a study that focused on compile time errors in Java[1]. The authors noted that compile time errors consume much of students' time and, contrary to common belief, the most common errors are equally difficult for *both* lower- and higher-ability students. The most common errors identified in this article were scoping errors (24%) and type errors (18%).

A more subtle approach to studying novice programmer errors is to observe how programmers respond to error messages arising from their mistakes. Such a study has been undertaken by Marceau *et al.* (2011a; 2011b) in the context of the FP language Scheme, concluding that error messages often fail to convey information regarding the cause of the error to the novice programmer. Further, the authors proposed a rubric for evaluating student responses to error messages. Our experience as teachers agrees with the somber conclusion of Marceau *et al.* in that error messages are often of little use to students. Our focus differs from Marceau *et al.* in that we wish to understand which kinds of errors students are likely to make and how these errors arise from the structure of the language. We use error messages only as the initial classification for the errors.

The closest reference to the present study is the one by Chambers *et al.* (2012), where student errors were collected using an observational method. Our study complements Chambers' study as we apply a computer-based analysis as the first step, similar to the studies by Jadud (2005), Fenwick *et al.* (2009), and Denny *et al.*

[1] The authors use the term syntax error to refer to scoping and type errors.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

(2012). Moreover, our method complements the work of Jadud, Fenwick, and Denny by incorporating a manual analysis step.

3 The course

The present study took place during an elective, eight-week course. The studied course initially had 88 registered students, and 55 students completed the first programming exercises. The majority of the participants, second- to fourth-year students, had already completed the early programming courses. During this course, we taught the basics of FP using Haskell, covering topics ranging from common FP constructs, such as recursion and folds (Hutton, 1999), to the use of a Hindley–Milner-like type system (Damas & Milner, 1982). The latter part of the course covers more advanced topics, such as type classes (Hall *et al*., 1996) and applicative functors (McBride & Paterson, 2008).

Our course emphasizes self-direction on the students' part, following the contemporary trends of blended learning (Garrison & Kanuka, 2004) and online education. The course material and exercises are online, but we support learning by having 6–8 hours of weekly, non-mandatory, contact teaching. Further, we allow students to work at their own pace through the provided material, within the general boundaries of the course schedule. This means that students can opt to spend more time on earlier topics and complete the course with partial credit, even if they cannot complete the later parts. The increased reliance on student self-regulation requires supporting students' self-directed learning (Isomöttönen & Tirronen, 2013). As part of this support, we provided students with an exercise assessment system that gives immediate automated feedback on the students' programs.

The automated feedback is provided through a web-based IDE consisting of a basic code editor and a Haskell interpreter. The system produces a report documenting the compilation errors, the results of the tests defined in the exercises, and the results of simple static analyses of program structure. The student answers are stored for grading purposes. The stored answers form an extensive database that can be used to analyze student performance.

4 Analysis procedure

When students submit their exercises for assessment, the system stores the submitted programs, the automated evaluation report, and a time stamp for later perusal. After the course, we took all the submitted exercises and compiled them, making a record of the compilation errors. We used the Glasgow Haskell Compiler (GHC), version 7.6.2, which was the most commonly used contemporary Haskell compiler at the time of the data collection. We also tested each answer that could be compiled for run-time errors. In total, we collected 4,843 student answers and the related errors. There were 509 exact duplicates among the answers, which were removed from the set. The duplicates are most likely due to the students resubmitting their answers again either accidentally or to ensure that they were observing the latest assessment results.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

Table 1. *Frequency of error messages during the course*

Name	Sessions	Attempts	Messages	Students	Exercises	Kind
Total	327	1,453	2,265	55	18	various
CouldntMatch	178	590	1,151	53	16	type
ParseError	138	326	326	51	16	syntax
NotInScope	109	217	384	46	16	syntax
NoInstance	23	51	75	15	8	type
NonExhaustive	22	63	63	15	7	run-time
TemplHaskell	17	31	31	16	9	syntax
OutOfRange	11	39	41	9	2	run-time
CantDeduce	7	17	17	7	1	type
InfiniteType	7	16	22	5	6	type
LacksBinding	7	11	11	7	7	syntax
NegativeExp	5	24	34	3	1	run-time
ConflicDef	5	15	17	5	3	syntax
ViewPatterns	5	7	7	5	3	syntax
HeadEmpty	5	7	7	4	3	run-time
StackOverflow	4	13	13	4	2	run-time
ScopedTypeVars	3	3	3	3	3	syntax
ArgsNum	3	3	3	2	2	syntax
TypeOperators	2	3	3	2	2	syntax
UndefinedArrayElem	1	11	11	1	1	run-time
NegativeIndex	1	6	6	1	1	run-time
RegexFail	1	5	5	1	1	syntax
LastEmpty	1	5	5	1	1	run-time
AmbiguousOcc	1	3	24	1	1	syntax
TooManyTypeArgs	1	2	2	1	1	syntax
CommentBracketMiss	1	1	1	1	1	syntax
IllegalDataDecl	1	1	1	1	1	syntax
TailEmpty	1	1	1	1	1	run-time
DuplicateSignatures	1	1	1	1	1	syntax

Answers that were returned anonymously were also removed, as we had observed that the students often used anonymous logins to the exercise system as a quick way of trying out unrelated programs. We then grouped the student answers into sessions, each of which represented a continuous slice of a single students' work, with no more than 40 minutes of idle time between the submissions. There were 890 sessions in total for the course. During these sessions, the students encountered 65 different types of errors, 26 of which originated from the compiler or the run-time system and 39 of which were related to failing to pass the tests included in the exercises.

The frequencies of errors emitted by the compiler and the run-time system are listed in Table 1. The first column in the table shows the labels given to different error messages (later referred to as the main error categories and detailed in Sections 6.1–6.3), the second column of the table gives the number of student sessions in which the error occurred, the third column lists the number of times a student submitted an exercise with the error for evaluation, and the fourth column gives the total number of error messages over all attempts, each of which can produce multiple

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

errors. The next two columns give the number of students who encountered the error and the number of exercises in which the error was emitted. The final column classifies the error types between type-, run-time, and syntax-errors depending on their usual cause. Thus, reading the *NoInstance* line, we can see that during the course there were 23 student sessions in which the *NoInstance* error appeared and a total of 51 answer submissions (attempts) in which the error was thrown at least once. The error was thrown 75 times and 15 students encountered this error, which appeared in eight different exercises. Lastly, the error was related to the type system.

We focus on errors that were encountered by more than one student during the course (the upper part of Table 1). The rest of the errors are accounted for the following sections when they, in combination, add significantly to the results. For each main error category, we manually studied all student sessions where the error occurred. During the investigation, we attempted to identify meaningful phenomena inside and across error case instances, and generated descriptive labels (codes). After the first round of coding, we picked the most relevant codes and coded the student answers again using the generated codes. In this phase, similarities between many codes became apparent and, we were able to reduce the number of codes while covering the relevant aspects in the data. The coding of the data was performed by the course instructor and one teaching assistant using a simple tool (see Figure 1) developed for viewing student practice sessions. Because the timestamps and earlier attempts are visible when using of this tool, this method of analyzing the submissions gives us a more comprehensive view of the students' thought processes compared to analyzing answers individually.

5 Limitations

Our study is based on data collected during a single course instance. There are some important limitations on the interpretations that can be drawn from it. Primarily, when looking at the student answers, we cannot be completely certain about the goal toward which the student was working. For example, we observed cases where a student had successfully completed an exercise and returned to tinker with it. We also found that students often submitted incomplete exercises, apparently just to see what would happen. In other cases, students might have had outside help either from the course supervisors or other students, which could have reduced the effort taken to complete the exercise.

Further, our means for data collection do not completely support all the analyses we wish to perform. For example, we did not track total student activity with the web-based exercises, and when measuring the time it took the students to complete a particular exercise, we could measure only the time interval between the first recorded attempt of submitting an exercise and the first successful attempt. This discounts the time used before the first submission and creates a difficulty in interpreting when the first attempt arrived on Monday morning and the exercise was completed late at Friday. For lack of a better measure, we divided sessions apart when there was an idle period lasting longer than 40 minutes and thus discounted all idle time longer than half an 40 minutes from the time data.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

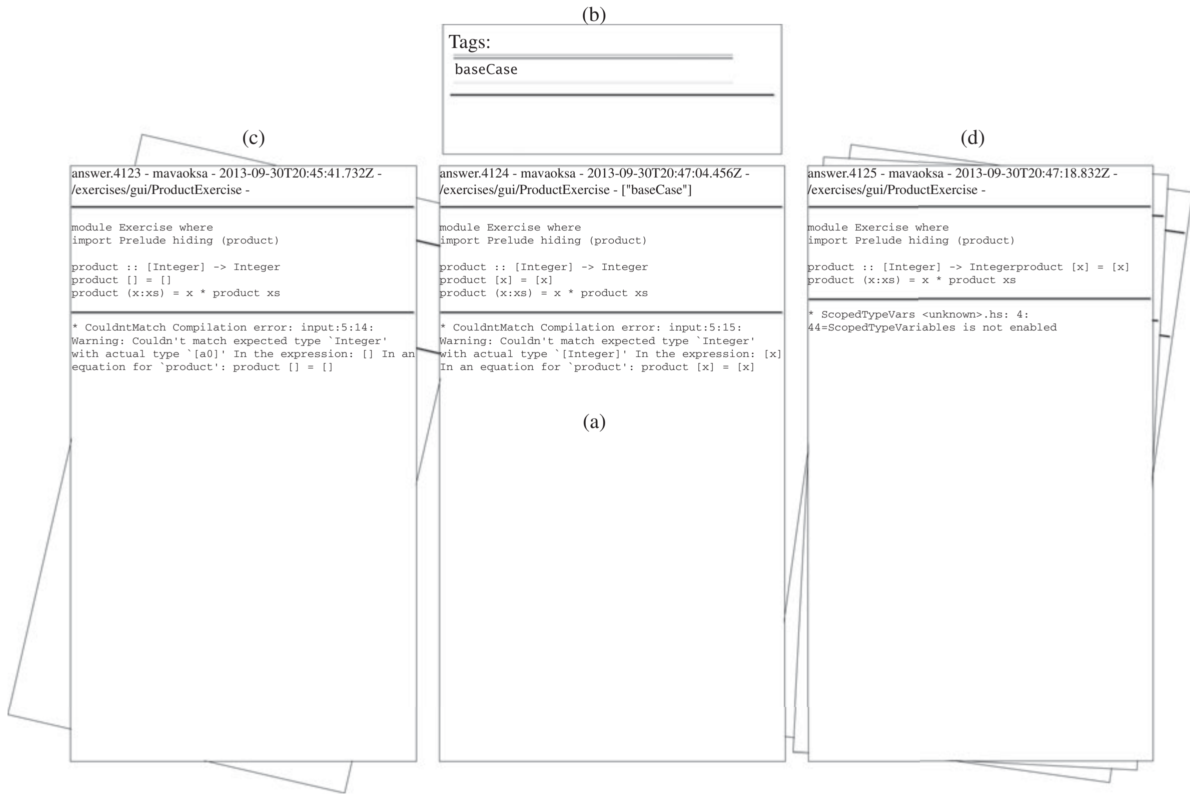

Fig. 1. The analysis tool for open coding. The view shows a single session, divided into three stacks. Each paper in a stack represents a single answer submission, showing when the answer was submitted, the answer itself, and the error messages associated with it. The stack in the middle (a) has the submission under analysis along with the related error messages, and the left (c) and right (d) stacks contain answers before and after the middle one, respectively. The box at the top (b) is for viewing and editing codes associated with the current submission. Users can navigate between submissions and sessions using the arrow keys.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

We must also note that the trends observed in this article are, by necessity, influenced by the set of exercises given to the students. For instance, we had an exercise that was specifically designed to challenge the students with the Haskell type system, and when we report that 69% of type errors were related to confusion between types, we must also note that 33% of such errors occurred with this specific exercise. Similarly, all errors where students indexed an array out of its bounds happened in the single exercise where arrays were introduced.

Similarly, failures to follow the exercise specifications are an interesting source of beginner difficulties, but they are harder to analyze without limiting the discussion to the context of a specific set of exercises. We have thus limited our data to errors arising from the compiler and the run-time system. This means that we do not explicitly inspect code that fails to follow our exercise specifications. Regardless, all such code has been part of the analysis process due to the inevitable compilation errors in such exercise submissions.

Finally, we must reiterate that the initial classification of the error types was done according to the types of error messages emitted by the GHC, version 7.6.2, and it is more than likely that various errors are reported differently with other compilers. This should not affect the manual analysis of the errors other than the structure in which the errors are presented in this article.

6 Results

An overview of the frequencies of the student errors is given in Figure 2. This table summarizes the number of student sessions within the main error categories arising from classifying compiler and run-time error messages. The main error category labels in the figure are detailed in Sections 6.1–6.3 and refer to specific kinds of errors emitted by the compiler and the run-time system. In the figure, the common error categories are annotated with the most common sources of the error and their ratio. These are indicated by a label and divider on the bar. In the figure, we use the code *Type* to refer to difficulties with the type system, *Syntax* to refer to conceptual difficulties with the Haskell syntax, and *Typo* to refer to errors resulting from simple mistakes such as misspelling a variable name. The codes *Constant* and *WrongNum* refer to mistakes with the Haskell numeric types, while the code *MissingCase* refers to missing cases in recursive functions.

The error distribution in Figure 2 is exponential: the majority of errors are captured by three main error message categories, and there is a long tail of less frequent errors, which only a few of the students encountered during the course. Leading this distribution are mistakes with types (*CouldntMatch*, *NoInstance*) and difficulties with syntax (*ParseError*, *NotInScope*). These errors are followed by run-time errors (*NonExhaustive*, *OutOfRange*) and a long tail of rare error types that arise from students triggering the more peculiar features of the compiler with obscure syntactic difficulties.

A closer inspection of the student answers reveals that difficulties with syntax are present in almost all main error categories. This observation holds even when simple mistakes, such as typographical errors (e.g., misspelled words, omitted symbols;

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

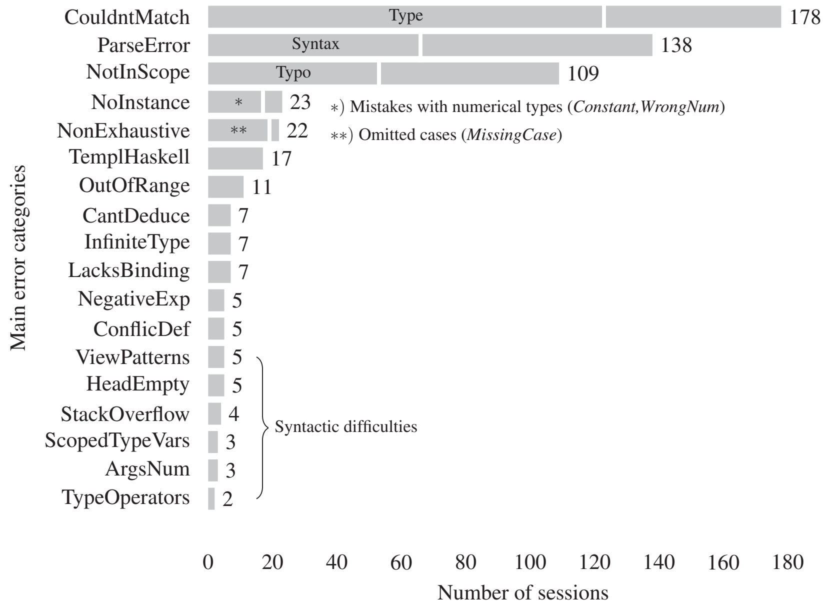

Fig. 2. Number of student work sessions where a given error occurred. See Sections 6.1–6.3 for explanations of the labels.

coded as *Typo*), are removed from the data, suggesting that the syntax of the language was poorly understood by the students. In accordance with the literature (Marceau *et al.*, 2011a), we observed that when encountering an error message, students often reacted inappropriately by either attempting to fix wrong parts of the program or by changing the program blindly without regard to the actual error. In our data, this problem is present in most of the error categories displayed in Figure 2, and particularly in the *CouldntMatch* category.

In the following sections, we present the results of the manual analysis of the error categories in Figure 2. For each category, we attempt to recategorize the different errors according to their observed root cause by manual inspection. We present the results grouped according to similarity between the categories, beginning with the error categories that relate to types, continuing with the ones relating to the syntax, and concluding with the run-time-related error categories. The subcategories obtained through the manual analysis overlap, meaning that a single student session can fall into several of the subcategories.

6.1 Error categories related to types

In this section, we analyze the error categories generally related to types: *Couldnt-Match*, *NoInstance*, *CantDeduce*, and *InfiniteType*. The code *CouldntMatch* represents errors where an operation expecting a specific type was given an incompatible type; *NoInstance* refers to errors where a function with a type class constraint is given arguments that do not satisfy the constraint; and *CantDeduce* refers to cases where the ex-

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

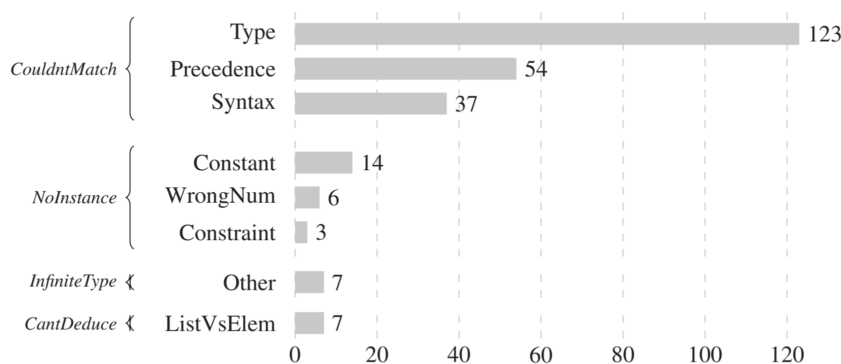

Fig. 3. Breakdown of the different causes of the type errors. The x-axis represents the number of student sessions.

pression might be well typed, but the compiler cannot infer the proper type automatically. Finally, *InfiniteType* labels situations where the inferred type of a term would be infinite, such as in `let x = ('x',x) in x :: (Char,(Char,(Char...`. The frequencies of these error categories are depicted in Figure 2, and they make up roughly 40% of all errors. They are present in 22% of all student exercise sessions.

6.1.1 CouldntMatch

When analyzing student sessions with the code *CouldntMatch*, we identified three main causes for this error message category: (1) actual type errors (*Type*), (2) errors caused by difficulties with precedence (*Precedence*), and (3) issues with syntax (*Syntax*). The number of student sessions where the difficulties fell into these subcategories is depicted in Figure 3.

In 69% of the sessions with *CouldntMatch* errors, the student passed an argument of the wrong type to a function or exhibited a lack of understanding of the type system. One-third of such sessions were concerned with an exercise specifically aimed to engage students with the type system, as depicted in Figure 4. Of the remaining sessions where the types did not match the most common error was mistaking an element inside a container, such as a list, with the bare element. Sessions where types were confused are labeled with the code *Type* in Figure 3.

Difficulties with precedence and the use of parentheses, labeled with the code *Precedence* in Figure 3, contributed to roughly 30% of the difficulties resulting in the *CouldntMatch* category of errors. The main cause for the high prevalence of errors related to precedence seems to be the ML-style function application syntax, where the function application is written as a juxtaposition without parentheses or punctuation. This causes student errors such as writing

```
fSort xs = concat sortBy comparing length group sort xs
```

in place of

```
fSort xs = concat (sortBy (comparing length) (group (sort xs))).
```

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

```
ex1 :: (a,b) -> a
ex1 = undefined
ex2 :: (a -> b) -> (b -> c) -> (a -> c)
ex2 = undefined
ex3 :: (a -> b) -> (c,a) -> b
ex3 = undefined
ex4 :: [a] -> [a] -> [a]
ex4 = undefined
ex5 :: (a, b) -> (a -> b -> c) -> (a,c)
ex5 = undefined
```

Fig. 4. The exercise designed to engage students with the type system. The student's goal was to give any well typed expression of the above types. This exercise appeared in context of other, more semantically meaningful, exercises on types and was intended to verify the essential understanding of types. This exercise was the leading cause of type errors during our course.

This issue is further exacerbated by the introduction of the function composition- (`.`) and the explicit application- (`$`) operators that are common in Haskell programs. These operators allow the programmer to write any of following definitions with the same effect as the previous definition:

```
fSort xs = concat . sortBy (comparing length) . group . sort
xs
fSort xs = (concat . sortBy (comparing length) . group . sort) xs
fSort    = concat . sortBy (comparing length) . group . sort
```

These operators seem to cause difficulties for many students, and we observed that blind testing was the dominant strategy employed in solving problems arising from these constructs. In addition, our subjective teaching experience was that a large number of students did not internalize the use of these operators during the entire course.

Finally, we observed that 20% of student sessions with the *CouldntMatch* errors were influenced by difficulties with other parts of the Haskell syntax (marked with the code *Syntax* in Figure 3). For example, errors that would be syntactic errors in many other languages are admitted as valid Haskell expressions due to Haskell's flexible syntax, but are rejected by its type checker. As a pathological example, one student repeatedly wrote

```
product (x:xs) = x (*) product xs,
```

which seems to be a misunderstanding of the syntax. However, this expression is parsed as a function application, where function `x` is passed three arguments (the `(*)`, the `product`, and the `xs`), causing a relatively involved type error to be presented to the student.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

6.1.2 NoInstance

The *NoInstance* errors occur when an ad hoc polymorphic function is used with values of type for which the function is not defined. The primary cause of this error in our experiment is Haskell's overloading of numerical constants. In 61% of these errors, a numerical constant was written in place of a term with a non-numerical type, such as a list or a function (see Figure 3, code *Constant*). In 26% of these error cases, the student had confused two numerical types or attempted to use an operator that works on a different numerical type (e.g., using division with integers; see Figure 3, code *WrongNum*). Almost all other *NoInstance* errors were due to the use of an ad hoc polymorphic function inside the student's own function definition without an appropriate type class constraint (see Figure 3, code *Constraint*), as demonstrated by the following function definition

```
group :: [a] -> [[a]]
group []      = []
group (x : y) = (x : z) : group w
                where (z, w) = span (x ==) y
```

The operator `==` is used without constraining the type `a` to be comparable (`Equation`).

6.1.3 InfiniteType and CantDeduce

The *InfiniteType* error is thrown when the student's definition is inferred to have an infinite type by the compiler. There were various reasons for these errors, including confusing a container, such as a tuple, with one of its elements. However, since the number of such errors was small and their causes varied, we could not discern a clear pattern, leading us to label their cause with the code *Other* in Figure 3. Similarly, the *CantDeduce* error messages are emitted when the compiler cannot automatically deduce the type for a given variable. In our data, this error is reported only when a student has confused lists (i.e., `[Int]`) with their elements (i.e., `Int`) in definitions that involve type class constraints, which makes this code a special case of the more common *CouldntMatch* type of errors (*ListVsElem* in Figure 3).

6.2 Error categories related to the syntax

This section reviews the error categories related to the syntax. The codes in question are the code *ParseError*, labeling situations where student program could not be parsed; *NotInScope* labeling cases where the compiler has encountered an undefined symbol; and the code *TemplHaskell* is given to errors that accidentally invoke the GHC metaprogramming system, which in our course are uniformly caused by writing expression outside of a definition. Finally, the *LacksBinding* errors are due to writing a type signature without the accompanying function definition, and the code *ConflicDef* arises when multiple definitions are given to the same variable. Finally, the code *ArgsNum* labels errors caused by defining a function by writing out different cases for it, but placing a different number of arguments for the cases.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

The remainder of error codes refer to accidentally triggering GHC extensions with the same name and are uniformly caused by difficulties with syntax.

The codes *ParseError* and *NotInScope* make up for one-quarter of all error messages, and a closer inspection reveals that the error codes *TemplHaskell*, *Lacks-Binding*, *ConflicDef*, *ViewPatterns*, *ScopedTypeVariables*, *ArgsNum*, *TypeOperators*, *RegexFail*, and *TooManyTypeArgs* are also caused by difficulties with syntax, while the code *CommentBracketMiss* is caused by omitting the closing brace of the comment block. We find that the majority of syntactic errors are caused by simple mistakes such as typos or missing parentheses. However, a significant part of these errors seem to be related to the misunderstanding of the semantics of Haskell. Altogether, similar to the studies by Jadud (2005) and Fenwick *et al.* (2009), we found that various syntax errors were also frequent in our data.

6.2.1 ParseError

The most common code related to syntactic errors in our data was labeled with the code *ParseError*. Such errors arise when the compiler is unable to parse the students' programs due to mismatched brackets, faulty indentation, misplaced =-operators, or a similar grammatical mistake. During the manual examination of the student sessions with this code, three distinct categories of errors surfaced: (1) faulty understanding of the syntax (*Syntax*), (2) missing or misplaced parentheses (*Parentheses*), and (3) typographical errors (*Typo*). The number of student sessions with these difficulties is depicted in Figure 5.

In this categorization, we distinguished typos and other small mistakes from apparent misunderstandings by examining the student sessions sequentially (see Figure 1 showing the analysis tool for explanation). If, for example, a student had written == instead of =, but corrected it mere seconds later, it is reasonable to assume that the error was not caused by a lack of knowledge. On the other hand, if an error was repeated in multiple parts of the program and it carried on to subsequent revisions of the same exercise without a clear indication that student was focused on correcting some other error, we assume that the student did not understand the syntax or the given error message.

A faulty understanding of the syntax (the code *Syntax*) seems to have affected roughly half of the student sessions labeled with parse errors. The gaps in student understanding include misusing the = -operator in the function definition, omitting commas from tuples, and using lambda-expression syntax incorrectly. The frequency of these errors exhibited a slight tendency to diminish toward the end of the course.

Missing parentheses, both "( )" and "[ ]", appeared in one-quarter of the parse error messages (see Figure 5, code *Parentheses*). One cause could be the rather deep nesting of expressions common to Haskell and functional languages in general. A noticeable subset of these errors consists of omitting parentheses when pattern matching function arguments, e.g., by writing `f x:xs = ...` instead of `f (x:xs) = ...`. The cause seems similar to the mistakes related to the issues of precedence and function application observed in Section 6.1; the students had general difficulties in recognizing operator and function application precedence.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

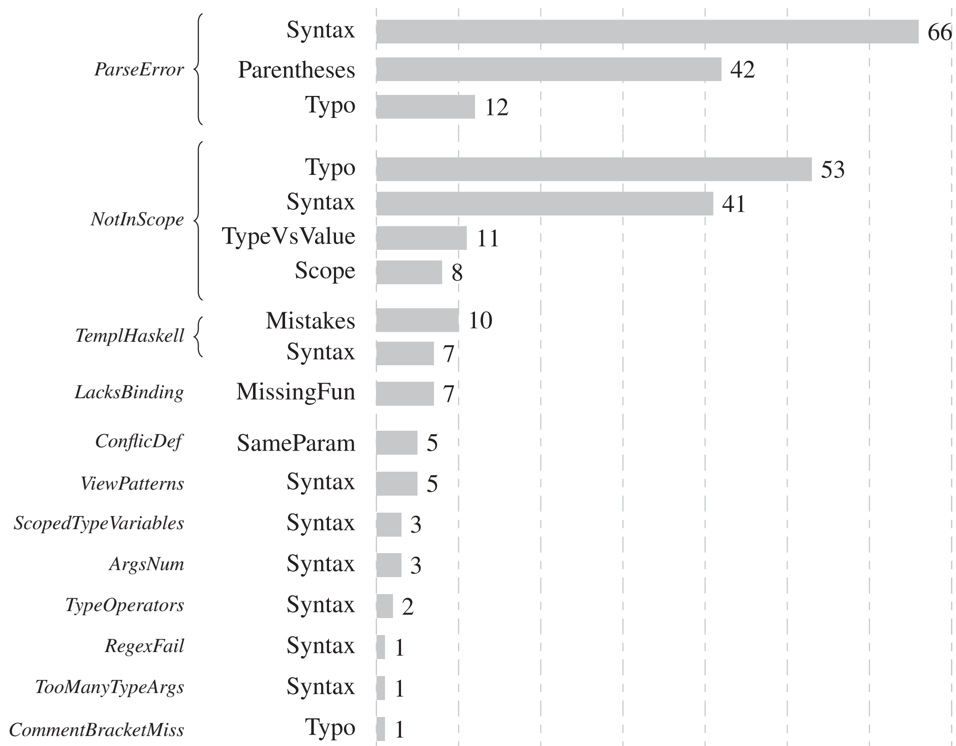

Fig. 5. Breakdown of the different causes of the syntax errors. The x-axis represents the number of student sessions.

When investigating the student answers with various typos, we did not discover any pattern that could be interpreted as specific to Haskell or FP in general (see Figure 5, code *Parentheses*). Similar to other languages, typos cause not only parse errors but also errors indicating undefined variables or mismatched types.

6.2.2 *NotInScope*

Errors labeled with the code *NotInScope* refer to situations where the compiler has encountered an undefined symbol. When reading through the student answers in sessions with this code, we discovered four distinct causes for these errors: (1) typos and other small mistakes (*Typo*), (2) faulty understanding of the syntax (*Syntax*), (3) confusing type-level variables with value-level variables (*TypeVsValue*), and (4) mistakes with the scoping rules of the language (*Scope*).

The number of student sessions influenced by these types of errors are shown in Figure 5. As expected, 49% of these errors can be explained by mistakes that imply no misconceptions on the part of the students and are usually quickly corrected.

However, 38% of student sessions with *NotInScope* errors were influenced by difficulties with the basics of the Haskell syntax. The most common difficulties were relatively minor issues, such as forgetting to capitalize the symbols `True` or `False` or using Java-style operator names such as `!=` in place of their Haskell

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

equivalents (i.e., /=), but there were a few more serious cases where the students exhibited a weak understanding of the structure of the language in general. For example, omitting the definition of function parameters was common, occurring in 22 student sessions. In some cases, this seemed due to students confusing the usual variable naming conventions with the actual syntax for passing arguments to the function. Some students seem to have assumed that the names `x` and `xs` automatically refer to the head and the tail of the argument or that variables `a` and `b` refer to the first and second arguments of the function. Examples of this error include `product = x * product xs` and `or = a || b`. In contrast to the cases of missing arguments that occurred for more experienced students, these errors persisted for many attempts and often include a host of other syntactic difficulties. The issues are labeled with *Syntax* in Figure 5.

Confusing type- and value-level variables is another relatively common mistake that causes *NotInScope* errors; these are labeled with the code *TypeVsValue* in Figure 5. In exercises where type variables were prominent, many students referred to the names of the type variables representing the type of the function parameter instead of the actual function parameter.

We also observed some difficulties where the error was actually due to the scoping rules of the language (see Figure 5, the code *Scope*). In isolated cases, students attempted to refer to names across function definitions. More commonly, however, the difficulties with the scoping rules were caused by the list comprehension syntax, where the variable introduction order is significant, in contrast to the rest of the language where it is not[2]. In some cases, students seemed to have assumed that the list comprehensions had constraint solving properties. For example, while generating a list of triples, a student specified two of the numbers and a condition that must hold for the third, with the apparent assumption that the program itself would figure out the proper value for the conditioned variable:

```
pythagoreanTriples :: Integer -> [(Integer,Integer,Integer)]
pythagoreanTriples n = [(a,b,c) | b <- [1..n], a <- [1..n]
                        ,a^2 + b^2 == c^2 ]
```

6.2.3 Other syntax-related error categories

The more infrequent errors in our data were also ultimately due to syntactic difficulties. For example, the code *TemplHaskell* is caused by naked top-level expressions. The error message for this mistake comes from the GHC's metaprogramming language extension, which, unfortunately, overrides the sensible default error message in our online exercises. About half of these errors are simple mistakes (see Figure 5, code *Mistakes*), such as forgetting to properly indent program code, while the remaining errors are due to difficulties with the Haskell syntax in general (see Figure 5, code *Syntax*).

[2] Discounting the `do` notation to which little attention was paid during the course.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

The errors with code *ConflicDef* arise from giving multiple contradictory definitions for the same variable (see Figure 5, code *SameParam*). In our data, this error is invariably exhibited by giving the same name to several parameters of the same function. In most cases, this seems to arise from blind testing, but in some cases, it seems that students assumed that Haskell patterns can be non-linear. For example, in an exercise requiring student to write program to remove consecutive duplicates, we saw the following attempt,

```
destutter []         = []
destutter (x:xs)     = x : destutter' xs x
destutter' (x:ls) x = destutter' ls x
destutter' (y:ls) _ = y : destutter' ls y
```

which only makes sense under non-linear pattern matching.

The remaining codes are all due to syntactic difficulties with Haskell. The *LacksBinding* error is due to a type signature without the accompanying function definition. This typically occurs when the function name is misspelled, or when the definition is accidentally commented out. Similarly, the *ArgsNum* error occurs when the student mistakenly places a different number of arguments in different cases of the same function. The codes *ScopedTypeVariables*, *ViewPatterns*, *TypeOperators*, and *RegexFail* refer to the inappropriate use of language extensions, but in practice are caused by difficulties with syntax, most often exhibiting some confusion between type- and value-level expressions. Finally, the code *TooManyTypeArgs* signifies giving too many arguments for a type constructor, which occurred due misunderstanding the syntax, and the code *CommentBracketMiss* indicates the omission of closing comment brace. These errors are labeled with the codes *Syntax* and *Typo* in Figure 5.

6.3 Error categories related to run-time

The codes *NonExhaustive* and *OutOfRange* (see Figure 2) represent the run-time errors that were caught by the GHC run-time system during the testing of the student programs by our exercise system. The code *OutOfRange* indicates errors stemming from indexing outside the bounds of a container such as an array. Similarly, the code *NonExhaustive* indicates a run-time error caused by pattern matching with a non-exhaustive set of patterns, none of which match the incoming data. Accessing containers outside of their bounds was relatively rare, occurring only in student sessions 13 during the course, whereas lacking an essential pattern in definitions was more common.

6.3.1 NonExhaustive, HeadEmpty, StackOverflow, and TailEmpty

The run-time errors *NonExhaustive*, *HeadEmpty*, *StackOverflow*, *TailEmpty* arise from similar mistakes. Non-exhaustive pattern matches make up the fourth most common code, *NonExhaustive*, which is a run-time error that is caused by partial functions. In our data, this error occurs most commonly when the student has forgotten to include a base case for a recursive function (see Figure 6, code *MissingCase*). In our data, missing base cases often arise from cases where there is more than one

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

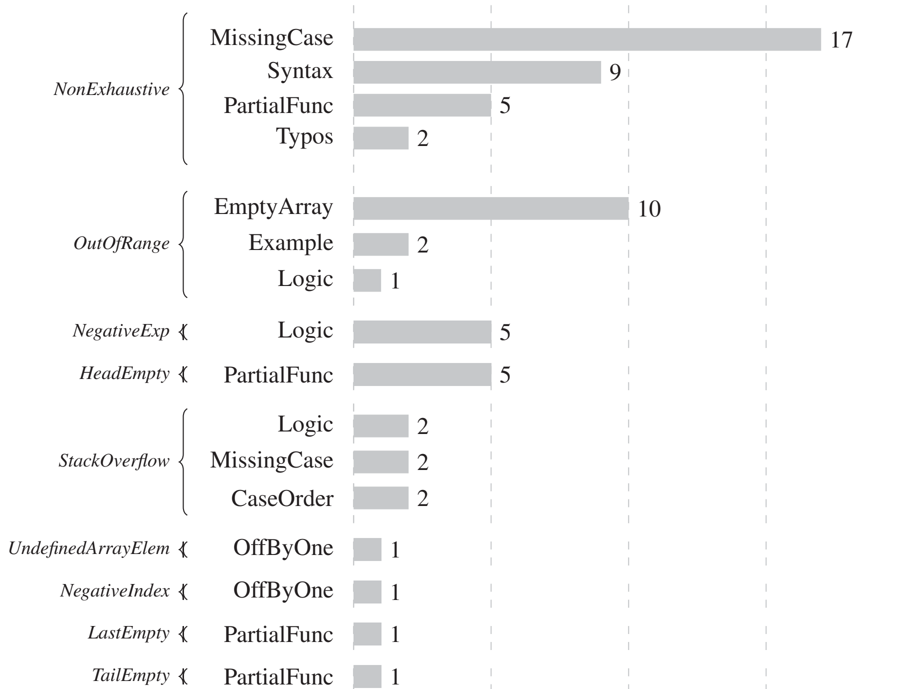

Fig. 6. Breakdown of the different causes of run-time errors. The x-axis represents the number of student sessions.

base case. For example, the pattern `[x]` is left out when cases `[]` and `x:y:xs` are included. This can either happen directly, or by first correctly matching `x:xs` and `[]` and then using a partial function, such as `head` on the remainder `xs`. In other words, the student does not know, or does not want to use, the nested pattern `x:y:xs`, making it harder to notice the missing base case.

Likewise, *StackOverflow* errors are sometimes relate to omitting the base case. Unlike *NonExhaustive* errors which terminate the program, some missing cases cause the program to enter infinite recursion, consuming all available resources before terminating. Furthermore, some run-time errors under the code *StackOverflow* result from the wrong order of an otherwise complete set of patterns. Haskell matches patterns from the top down, and sometimes students had an all capturing general case pattern that shadowed the necessary base cases. Although this pattern occurs in conjunction with the code *StackOverflow* in our data, it is a viable cause for other run-time errors as well.

Another common difficulty, restricted to a limited number of students, was to completely omit the base case. A mundane explanation for this is that the students started from the recursive case and, being absorbed by the work, simply forgot to complete the assignment. Overall, missing cases make up 53% of *NonExhaustive* errors, while the rest are difficulties with syntax.

Missing base cases have been studied in the CS1 literature (Kahney, 1983; Segal, 1994; Haberman & Averbuch, 2002) where they are often attributed to faulty mental models or misconceptions caused by analogies used in teaching. In our case,

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

we did not detect any general misconceived mental model explaining the difficulties; instead, we saw difficulties in generalizing the language constructs and noticing the less obvious base cases.

The use of partial functions, coded with as *PartialFunc* and also reported under codes *HeadEmpty*, *NonExhaustive*, and *TailEmpty*, causes errors outside of recursive definitions. In these cases, students failed to account for edge cases when using partial functions from Haskell standard libraries.

Syntactic difficulties (see Figure 6, code *Syntax*) were also a prominent cause of run-time errors. For example, 19% of the *NonExhaustive* error cases were caused by pattern matching a list into the head and tail without a real need to do so. Our course has more examples where lists are pattern matched than those where they are used as is. This suggests that the cause of the spurious pattern match is that most examples concerning lists use pattern matching and the students simply follow the convention without considering why. This can be linked to the misplaced knowledge concept described by Perkins & Martin (1986). In addition, we saw an example in which a student had refactored code by moving the base case of a function to a locally defined auxiliary function, leaving the program with a spurious pattern match.

Many other kinds of syntactic mistakes led to executable programs that failed with a run-time error (Figure 6, code *Syntax*). For example, one student wrote

```
destutter :: Eq a => [a] -> [a]
destutter [] = []
destrutter (x:xs)
    | x == head xs = destrutter xs
    | otherwise = x:(destrutter xs)
```

where the misspelling of `destutter` led to two, very partial, function definitions.

6.3.2 OutOfRange, UndefinedArrayElem, and NegativeIndex

Errors that occur when an array is indexed outside its bounds are labeled with the code *OutOfRange*, while the codes *UndefinedArrayElem* and *NegativeIndex* are special cases of this error. These errors were relatively common in the array processing exercise in the course, where students were required to check whether a given array contained a palindrome. The most common cause of *OutOfRange* errors was failing to check whether the array was empty (see Figure 6, code *EmptyArray*). We conjecture that the failure to recognize empty arrays was compounded by the convention of reporting the array bounds instead of the array length in the array library we used (`Data.Array`). Students might have mistaken the upper bound with the length. Other causes of this error were arithmetic mistakes (Figure 6, code *Logic*) and writing the program so that it worked only for the example case presented in the exercise assignment (see Figure 6, code *Example*). Finally, the code *UndefinedArrayElem* was caused by off-by-one errors (*OffByOne*) while constructing an array while the *NegativeIndex* error was caused by array underflows in other situations (*OffByOne*).

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

6.3.3 *NegativeExp*

Last, we saw many errors reporting the computation of negative exponents for integral types. All instances of this error arose from an exercise where the students were asked to convert a list of bits to an integer and vice versa. In all cases, these errors were due to mistakes in program logic, most commonly occurring when decrementing the exponent with each recursive step instead of incrementing it. Although this exercise would have been easy to complete by following the intended binary representation of least significant bit first, many students wrote the recursion in order of most significant bit first, requiring them to carry the exponent as auxiliary value throughout the computation. This order results in recursive structure that is different from that of the structural recursion exercises along with which this exercise appeared with, which can have had an effect on the student difficulties at this point.

6.4 Difficulty of the errors

The impact of the various errors on student productivity is depicted in Table 2. This table shows an approximation of how long it took the students to excise certain kinds of errors from their programs and was derived by measuring from the time when the student first submitted an answer with a given type of error message to the time when a submission without this type of error was received. This measuring scheme implies that if a student had multiple consecutive errors of the same type, their time span is summed. More critically, the measurement does not contain information about the persistence of individual errors, only about the main categories of errors in the program. As explained by Marceau *et al.* (2011a), deducing whether a programming error was fixed is not possible simply by observing that the particular error message is no longer present. The table can only estimate the difficulty of problems that are characterized by the appearance of certain types of errors.

The effect of the different exercises must also be taken into account when inspecting the table. For example, *ViewPatterns* errors occurred mostly in exercises dealing with type system, which was observed to be a difficult topic for our students. Similarly, the *NegativeExp* occurred solely in a recursive exercise requiring students to convert natural numbers between binary and decimal representations. This exercise was also observed as difficult for the students.

The table hints that, in total, type errors consumed most of the students' time, followed by syntax errors. Run-time errors, although individually difficult to solve, occurred less frequently and thus took up less time in total.

The reasons for the abundance of type errors are likely manyfold: the flexible syntax of Haskell results in many different mistakes to be reported as type errors, the difficulties in understanding the Hindley–Milner style type system result in more type-related misconceptions, and the way the type errors are reported may cause them to be more difficult to interpret.

In total, syntax errors consumed much of the students' time, but seemed individually fast to correct. The obvious exceptions are *ViewPatterns*, *ArgsNum*, and

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

Table 2. *Error survival rates. The first four columns specify the percentage of errors solved in given amount of time. The fifth column states how many times the error (re)appeared, and the last two columns present the total time the error remained in student's code and the average time for the errors to disappear*

		1 min	5 min	15 min	30 min	Total	Total Time	Average Time
type	CouldntMatch	40%	77%	94%	99%	228	15h 17m	4m
	CantDeduce	73%	82%	91%	100%	11	41m	4m
	InfiniteType	60%	80%	100%	100%	5	13m	3m
	NoInstance	37%	85%	100%	100%	27	1h 1m	2m
runtime	LastEmpty	0%	0%	0%	0%	1	40m	40m
	NegativeExp	20%	40%	80%	80%	5	48m	10m
	UndefinedArrayElem	50%	50%	100%	100%	2	12m	6m
	StackOverflow	25%	25%	100%	100%	4	19m	5m
	OutOfRange	33%	67%	94%	100%	18	1h 21m	5m
	IndexTooLarge	0%	50%	100%	100%	2	8m	4m
	NonExhaustive	27%	77%	92%	100%	26	1h 42m	4m
	HeadEmpty	50%	100%	100%	100%	4	5m	1m
	NegativeIndex	75%	100%	100%	100%	4	4m	1m
	TailEmpty	100%	100%	100%	100%	1	1m	1m
syntax	ViewPatterns	17%	50%	67%	100%	6	53m	9m
	ArgsNum	67%	67%	67%	100%	3	19m	6m
	RegexFail	0%	50%	100%	100%	2	11m	6m
	ParseError	60%	86%	98%	100%	173	6h 21m	2m
	LacksBinding	62%	88%	100%	100%	8	17m	2m
	NotInScope	67%	91%	98%	100%	127	3h 52m	2m
	ConflicDef	75%	88%	100%	100%	8	14m	2m
	ScopedTypeVars	0%	100%	100%	100%	2	3m	2m
	TemplHaskell	64%	100%	100%	100%	14	17m	1m
	DuplicateSignatures	0%	100%	100%	100%	1	1m	1m
	TypeOperators	100%	100%	100%	100%	2	1m	0m
	AmbiguousOcc	100%	100%	100%	100%	3	1m	0m
	IllegalDataDecl	100%	100%	100%	100%	1	0m	0m
	CommentBracketMiss	100%	100%	100%	100%	1	0m	0m

RegexFail, which rank among the most time-consuming error messages. All these errors are relatively uncommon and likely to occur in cases where the student's code is in disarray. For example, the code *ViewPatterns* occurs most readily when student confuses type- and value-level expressions.

Run-time errors consumed the least total time, but there are some common run-time errors, such as *OutOfRange*, *StackOverflow*, and *NegativeExp*, that consume significantly more time than other common errors even though they are caught by the tests in our web exercise system as quickly as compile time errors are. In the case of the code *NonExhaustive*, this difficulty might be due to difficulties in understanding the concepts of recursion and pattern matching rather than difficulties in identifying the cause of the run-time error. The code *OutOfRange* occurs only in exercises related to array processing. Arrays are often regarded as a difficult topic by CS

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

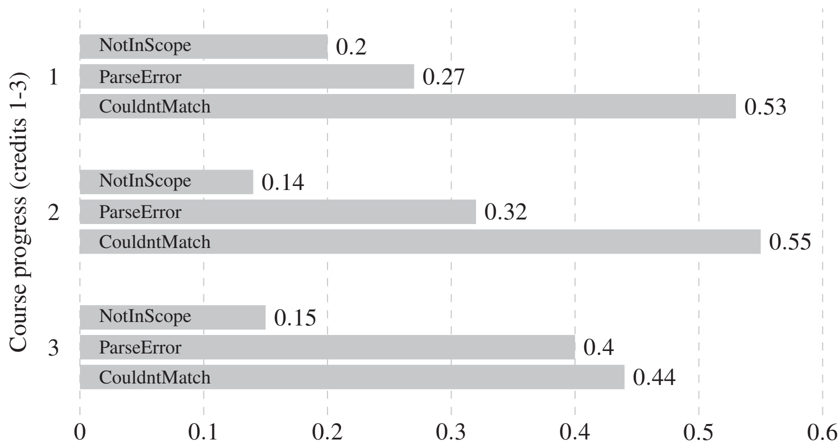

Fig. 7. Evolution of error frequencies respective to course progress during the first three fifths of the course. The unit of the x-axis is the ratio of the specific error messages respective to all error messages.

educators, and this domain difficulty might explain the relatively long time it took students to resolve these errors.

6.5 *Frequencies of errors over time*

Figure 7 displays the relative error frequencies during the first three sections of the course (the last two sections are omitted as only a few students completed them on time). This figure shows that the the three main categories of errors are approximately constant over the course. We observe a slight decrease in *NotInScope* errors after the first section of the course and a significant increase in the *ParseError* category. Syntactic structures are gradually introduced, and each section of the course contains more syntax and more involved exercises than the preceding one, which makes the increase of syntactic difficulties natural. In the case of *NotInScope* errors, all structures that relate to variable scope are given in the first section with the exception of the list comprehensions that appear in the third part of the course.

We can conclude that measuring the evolution of error frequencies over time during an ordinary programming course does not seem to offer an easy way to observe which error categories are more persistent than others. The course topics and the expected student difficulties change during the course. To investigate this effect, the studied course would need to incorporate a plateau period during which no new topics or new type of exercises would be introduced and during which the measurement could take place.

7 Discussion

In this paper, we studied student exercise answers and applied both computer and manual analyses to them. We identified several common, low-level mistakes that

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

students make. Several of the mistakes seem to be directly induced by the Haskell language, whereas others are more indicative of difficulties in teaching.

7.1 Language design issues

Of the specific issues we observed, the difficulties with interplay of precedence, associativity, and juxtaposition as function application syntax seem surprisingly serious. As a specific example of a syntactic problem arising from the Haskell base library, the function composition `(.)` and explicit application `($)` operators were common causes for syntactic difficulties. Therefore, although composition and application operators offer advanced programmers useful flexibility, their usefulness should be questioned at the beginner level. Furthermore, when designing a beginner friendly language, we suggest careful consideration whether to adopt juxtaposition as the syntax of function application in the first place.

In some cases, syntactic problems seem to obscure otherwise clear concepts. For example, Chakravarty & Keller (2004) advise against teaching partial application in early courses due to the mathematical proficiency required. Based on our data, we agree with this advice, but with a different rationale. We did not observe difficulties with the concept of partial application in our data nor in our teaching. However, we observed many difficulties with the syntax it gives rise to in Haskell. We posit that including the partial application in languages intended for teaching beginners does not inherently have a high cost, but doing so would benefit from a different syntax.

In addition to syntax, we also found that the Haskell standard library (Prelude) was responsible for several student errors, such as inducing run-time errors due to use of partial functions such as `head` or `tail`. In our course, nearly all cases where these functions were used could have been rewritten to use pattern matching, the completeness of which can be checked by the compiler. We suggests reducing the number of partial functions in default libraries in favor of pattern matching. Based on the observed errors, this change could be helpful in directing beginner students toward better habits. We further posit that expert programmers are also likely to benefit from making partiality explicit: the exercises in our course are low level and prone to use of partial functions. On the other hand, "real-world" program code, which makes use of higher-level functions, has perhaps fewer places where functions such as `head` or `tail` can be used effectively. Unlike in the course exercises, which include rigorous testing, catching run-time errors in real-world code can be much harder and costlier, giving expert programmers one more reason to avoid them.

Issues related to the misuse of pattern matching seem to be due to either difficulties in nesting the patterns or accidentally omitting essential cases. Unfortunately, the compiler option to warn about missing cases is not switched on by default in GHC, and our data suggest that missing cases are time-consuming for the students. From the point of view of a teaching language, we see no downside to enabling the related compiler warning message or even promoting it to an error. Furthermore, it seems that the related run-time error messages could be easily improved by indicating which datum was the one that was not matched.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

Three common error message types cover the vast majority of errors encountered by beginners. We doubt that there is much that can be done to improve syntax errors or how the compiler refers to misspelled variables, but there are things that should be done regarding type errors. The structure of Haskell is very flexible, and the majority of programmer errors are reported as type errors claiming some type to be incompatible with some other type. Type errors arise when the student has mistaken an `Int` for a `String` or forgotten to pass one argument to a function. As such, the largest bin of compiler error messages contains wildly different student errors, all which are reported uniformly. We suggest that differentiating the largest classes of error messages further would be helpful for beginners as well as expert programmers. There is a large body of work on improving type error messages in the ML family of languages (Heeren, 2005; Hage & Heeren, 2007; Lerner *et al*., 2007), as well as improving error messages in general (Marceau *et al*., 2011b). This advice and research seems often to be neglected in practice.

Before undertaking this study, we heard various claims of what is difficult for beginners. Several of these claims failed to be supported in our data. For example, the difficulties with recursion proposed by Segal (1994), such as mistaking base case for a stopping condition, were not observed.

Similarly, lazy evaluation is sometimes perceived as a difficult topic for beginners. Regardless, our experience was that laziness did not cause significant difficulties in the early part of the course. On the contrary, the non-strict semantics of Haskell made it easier for us to teach concepts by demonstrating the steps of the computation. Contrasting this to the early literature, Anderson *et al*. (1988) claim that evaluating recursion mentally is very hard. As an example, Anderson claims that arithmetic expressions such as `4 * (3 - 2) * (5 + 7)` are not evaluated by progressing recursively from top-down according to the syntactic structure of the formula. Instead, the authors claim that it is natural for the human mind to progress (iteratively) by computing the simplest subexpressions first. Contrary to claim by Anderson *et al*., changing the evaluation order does not make the computation iterative; any such order of computations is permissible with non-strict semantics and the evaluation of such expressions can proceed in any order perceived as natural. Our teaching experiences agree with Anderson that the "simplest-first" order can be much easier for students as it enables us to demonstrate the evaluation of programs in a natural way (Tirronen & Isomöttönen, In Press). We also found non-strictness to be helpful when demonstrating more complex recursive functions by writing out their computation. Regardless of the early benefits of non-strict semantics, building explicitly lazy functions seemed to cause difficulties for the students later on. Moreover, discussions of the efficiency of basic functions such as `foldl` require discussion of normal forms and evaluation order, which we assume to be difficult for beginners.

Even though programming folklore often rates array out of bounds style errors in the top three most common errors (Leblanc & Fischer, 1982), we observed rather few cases of this during our course. Haskell, and modern languages in general, seems to present an improvement over many mainstream languages on this front since numerically indexed containers are rarely needed and because most containers

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

are usually either accessed by functions, such as `map` or `sum`, which do not permit such errors.

7.2 Issues in teaching functional programming

Our study focused on the low-level mistakes that occur during the process of solving programming exercises, which seems to leave out difficulties in areas such as problem solving. Our data (Figure 7) show that issues with syntax tend to persist through the course and our subjective teacher experience was that syntactic issues increased the cognitive burden of students by impairing their problem-solving skills. Our findings differ from those of the earlier work by Soloway & Ehrlich (1984) who claim that misconceptions about language constructs, such as loops and conditionals, are not the main cause of difficulties. Instead, similar to Denny *et al.* (2012), we find that there are persistent low-level errors that consume much of the students' energy. This is possibly due to the simpler language used in the study done by Soloway *et al.*

Due to the surprising amount of difficulties with function application syntax, we urge instructors who make use of ML-like languages to experiment with the use of a single parameter function with tuple arguments in place of multiparameter functions, i.e., by writing

```
sortBy(comparing(length), group(xs))
```

in place of the more Haskell-like

```
sortBy (comparing length) (group xs).
```

This approach could make the distinction between functions and their arguments more apparent and could eliminate many of the errors that arise from precedence, parentheses, and misusing the operators `(.)` and `($)`, which were common causes of syntactic difficulties. Therefore, although the use of composition and application operators can provide useful flexibility to advanced programmers, we recommend that instructors of beginner students avoid them entirely until the students have achieved sufficient proficiency with the language.

Further, to state the obvious, errors with precedence can be reduced by avoiding nested expressions in favor of naming the subexpressions; the problematic definition

```
fSort xs = concat (sortBy (comparing length) (group (sort xs)))
```

can be written, with some loss of conciseness and elegance, as

```
fSort xs = let
        sorted   = sort xs
        grouped  = group sorted
        resorted = sortBy comp grouped
        comp     = comparing length
       in concat resorted
```

where there is no possibility of making a mistake with parentheses. It can, however, be contested whether naming subexpressions gives rise to a different kind of cognitive load.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

Other causes of syntactic difficulties seemed to have had a smaller impact in our course, and we theorize that, for the most part, they are caused by misplaced and conglomerated knowledge of Haskell syntax and usage patterns. This suggests that teaching efforts should be spent to illustrating when each language construct should, or should not, be used. For example, the majority of our list processing examples included pattern matching, which led some students to apply pattern matching in situations where there were lists but no need to destruct them in any way.

Similarly to the literature (Clack & Myers, 1995; Heeren *et al.*, 2003; Hage & Keeken, 2006), the Haskell type system was found to cause difficulties, and it is clear to us that the teaching of types requires a more effective strategy than presenting examples and hoping that the students "catch on" (Ruehr, 2008; Tirronen & Isomöttönen, In Press). Many of the issues with types are linked to issues with syntax, since understanding Haskell type expressions depends on a proper understanding of associativity and precedence, similar to findings by Joosten *et al.* (1993).

We also conjecture that number of run-time errors could be reduced with didactic techniques. Even though our web-based exercises report run-time errors with the same mechanism as any other error, they still consumed a significant amount of student time. The involved error messages might be improved by enabling explicit warnings about non-exhaustive patterns, and similar to Felleisen *et al.* (2001), it might be beneficial to mandate students to follow a design recipe in which different cases of input data are explicitly specified before writing code.

Student difficulties often manifest themselves through error messages, and much work has been done to make Haskell error messages more novice-friendly (Heeren, 2005). Based on our observations, the three most common error messages cover the vast majority of the errors the students faced during the course. Naturally, the most advisable solution would be to improve the compiler itself to emit better messages, but this is challenging enough to be out of reach for most instructors. In cases such as ours, where many early exercises are completed using a custom (web-based) system, it is feasible to customize the error messages displayed to the students, more specifically by attempting to differentiate the various causes of the three dominant error types.

Another possible approach in lessening the contemporary difficulties with error messages could be to introduce the students to different causes for the common error messages early on. In contrast to modifying the error messages for the needs of a specific course, this approach has the benefit of allowing the students to work with the original compiler messages, which is what the students will inevitably face when working outside of the course tool support while still alleviating the difficulties during the course.

8 Conclusions

The goal of this study was to understand the low-level errors made by beginner Haskell programmers and to study how student errors are reported in compiler error messages. Methodologically, we found that it is not possible to form a comprehensive

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

view of the various problems in students' mindsets simply by inspecting the ratios of error messages. For example, a faulty operator usage might have caused a type error, but the cause of that error might be a misunderstanding of the syntax, an error with brackets or precedence, or some other cause. A student might also be blindly throwing operators around and hoping that some attempt will work. By viewing the errors in the context of the session where they appeared, we were able to get an extensive view of the issues behind the errors. A sequential examination of student submissions revealed how the student handled the error, in some cases how the error first appeared, and in some cases how the error persisted in different forms across attempts.

Our study revealed several causes of student errors that stem from the language design. These issues include adopting a too flexible syntax, which causes difficulties with precedence, function application, and deeply nested statements. Regardless of being "just syntax", these difficulties persist throughout the course. Another language design issue regarding beginner students is that the standard libraries make using partial functions too attractive choice for the students, causing run-time errors. Finally, as documented by Chambers *et al.* (2012), type errors were very common.

One important finding is that majority of student errors are reported with three different compiler error messages, which are not well understood by students. These error messages conglomerate errors from many different causes, further reducing their usefulness for the beginner student.

We also concluded that many difficulties were due to instruction. Starting directly from a functional mindset and presenting deeply nested expressions is not a fruitful approach to teaching FP. Furthermore, we believe that the Haskell type system must be taught both carefully and explicitly early on: most beginner errors manifest as type errors in Haskell.

We posit that contrary to popular opinion, syntax does matter and point the reader toward studies such as the one by Stefik & Siebert (2013), which presents convincing results that different choices of syntax, even within a single language paradigm, can have a major effect on the success of beginner programmers.

References

Anderson, J. R., Pirolli, P. & Farrell, R. (1988) Learning to program recursive functions. In *The Nature of Expertise*, Robert, G., Chi, M. T. H. & Farr, M. J. (eds), Hillsdale: Psychology Press; ISBN-13: 978-0805804041, pp. 153–184.

Bieniusa, A., Degen, M., Heidegger, P., Thiemann, P., Wehr, S., Gasbichler, M., Crestani, M., Klaeren, H., Knauel, E. & Sperber, M. (2008) HtDP and DMdA in the battlefield. *Functional and Declarative Programming in Education*. Victoria, BC, Canada.

Blanco, J., Losano, L., Aguirre, N., Novaira, M. M., Permigiani, S. & Scilingo, G. (2009) An introductory course on programming based on formal specification and program calculation. *ACM SIGCSE Bull.* **41**(2), 31–37.

Bonar, J. & Soloway, E. (1985) Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Comput. Interact.* **1**(2), 133–161.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

Brown, N. C. C. & Altadmri, A. (2014) Investigating novice programming mistakes: Educator beliefs versus student data. In Proceedings of the 10th annual conference on International Computing Education Research. University of Glasgow, Glasgow, Scotland: ACM.

Chakravarty, M. M. T. & Keller, G. (2004) The risks and benefits of teaching purely functional programming in first year. *J. Funct.Program.* **14**(1), 113–123.

Chambers, C., Chen, S., Le, D. & Scaffidi, C. (2012) The function, and dysfunction, of information sources in learning functional programming. *J. Comput. Sci. Colleges* **28**(1), 220–226.

Clack, C. & Myers, C. (1995) The Dys-functional student. In *Functional Programming Languages in Education*, P. Hartel & R. Plasmeijer (eds), Lecture Notes in Computer Science, vol. 1022. Berlin/Heidelberg: Springer, pp. 289–309.

Damas, L. & Milner, R. (1982) Principal type-schemes for functional programs. In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Albuquerque, New Mexico, USA: ACM Press, ISBN 0-89791-065-6, pp. 207–212.

Denny, P., Luxton-Reilly, A. & Tempero, E. (2012) All syntax errors are not equal. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education. Haifa, Israel: ACM, pp. 75–80.

Farchi, E., Nir, Y. & Ur, S. (2003) Concurrent bug patterns and how to test them. In Proceedings of Parallel and Distributed Processing Symposium, International, Nice, France, 2003. IEEE, p. 7.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2001) *How to Design Programs.* Cambridge: MIT Press.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2004) The structure and interpretation of the computer science curriculum. *J. Funct. Program.* **14**(4), 365–378.

Fenwick Jr, J. B., Norris, C., Barry, F. E., Rountree, J., Spicer, C. J. & Cheek, S. D. (2009) Another look at the behaviors of novice programmers. *ACM SIGCSE Bull.* **41**(1), 296–300.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. & Felleisen, M. (2002) Drscheme: A programming environment for scheme. *J. Funct. Program.* **12**(02), 159–182.

Garrison, D. R. & Kanuka, H. (2004) Blended learning: Uncovering its transformative potential in higher education. *Internet Higher Educ.* **7**(2), 95–105.

Haberman, B. & Averbuch, H. (2002) The case of base cases: Why are they so difficult to recognize? student difficulties with recursion. In ACM SIGCSE Bulletin, vol. 34. ACM, pp. 84–88.

Hage, J. & Keeken, P. (2006) *Mining for Helium. Technical report UU-CS.*

Hage, J. & Heeren, B. (2007) Heuristics for type error discovery and recovery. In *Implementation and Application of Functional Languages*, Springer, pp. 199–216.

Hall, C. V., Hammond, K., Peyton Jones, S. & Wadler, P. (1996) Type classes in Haskell. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **18**(2), 109–138.

Heeren, B. J. (2005) Top Quality Type Error Messages, 2005/9/20, IPA Dissertation Series, Utrecht University.

Heeren, B., Leijen, D. & van IJzendoorn, A. (2003) Helium, for learning Haskell. In Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell. Uppsala, Sweden: ACM, pp. 62–71.

Hutton, G. (1999) A tutorial on the universality and expressiveness of fold. *J. Funct. Program.* **9**(4), 355–372.

Isomöttönen, V. & Tirronen, V. (2013) Teaching programming by emphasizing self-direction: How did students react to active role required of them? *Trans.Comput. Educ.* **13**(2), 6:1–6:21.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

Jadud, M. C. (2005) A first look at novice compilation behaviour using BlueJ. *Comput. Sci. Educ.* **15**(1), 25–40.

Joosten, S., Berg, K. & Hoeven, G. V. D. (1993) Teaching functional programming to first-year students. *J. Funct. Program.* **3**(1), 49–65.

Kahney, H. (1983) What do novice programmers know about recursion. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. Boston, MA, USA: ACM, pp. 235–239.

Keravnou, E. (1995) Introducing computer science undergraduates to principles of programming through a functional language. In Proceedings of the 1st International Symposium on Functional Programming Languages in Education. (FPLE '95), LNCS 1022, Nijmegen, The Netherlands: Springer-Verlag, ISBN-13: 978-3540606758, pp. 15–34.

Kinnunen, P. & Malmi, L. (2006) Why students drop out CS1 course? In Proceedings of the 2nd International Workshop on Computing Education Research. University of Kent, Canterbury, UK. ICER '06. New York, USA: ACM, pp. 97–108.

Lahtinen, E., Ala-Mutka, K. & Järvinen, H. M. (2005) A study of the difficulties of novice programmers. *SIGCSE Bull.* **37**(3), 14–18.

Leblanc, R. J. & Fischer, C. N. (1982) A case study of run-time errors in Pascal programs. *Softw.: Pract. Exper.* **12**(9), 825–834.

Lerner, B. S., Flower, M., Grossman, D. & Chambers, C. (2007) Searching for type-error messages. In ACM SIGPLAN Notices, vol. 42. ACM, pp. 425–434.

Lewandowski, G. (2003) Using process journals to gain qualitative understanding of beginning programmers. *J. Comput. Sci. Colleges* **19**(1), 299–310.

Lu, S., Park, S., Seo, E. & Zhou, Y. (2008) Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In ACM SIGPLAN Notices, vol. 43. ACM, pp. 329–339.

Ma, L., Ferguson, J., Roper, M. & Wood, M. (2011) Investigating and improving the models of programming concepts held by novice programmers. *Comput. Sci. Educ.* **21**(1), 57–80.

Marceau, G., Fisler, K. & Krishnamurthi, S. (2011a) Measuring the effectiveness of error messages designed for novice programmers. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education. ACM, pp. 499–504.

Marceau, G., Fisler, K. & Krishnamurthi, S. (2011b) Mind your language: On novices' interactions with error messages. In Proceedings of the 10th SIGPLAN. Tallinn, Estonia September 26–28, 2005, Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, pp. 3–18.

McBride, C. & Paterson, R. (2008) Functional pearl: Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Morazán, M. T. (2011) Functional video games in the CS1 classroom. In *Trends in Functional Programming*, May 16–18, 2011, Madrid, Spain: Springer, pp. 166–183.

Morazán, M. T. (2012) Functional video games in CS1 II. In *Trends in Functional Programming*, Springer, University of St Andrews, Scotland, UK, pp. 146–162.

Pane, J. F., Ratanamahatana, C. A. & Myers, B. A. (2001) Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Human-Comput. Stud.* **54**(2), 237–264.

Pea, R. D. (1986) Language-independent conceptual "bugs" in novice programming. *J. Educ. Comput. Res.* **2**(1), 25–36.

Perkins, D. N. & Martin, F. (1986) Fragile knowledge and neglected strategies in novice programmers. In Proceedings of 1st Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers, Washington, DC, USA, pp. 213–229.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press

Ruehr, F. (2008) Tips on teaching types and functions. In Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education, September 21, 2008, Victoria, British Columbia, Canada: ACM, pp. 79–90.

Segal, J. (1994) Empirical studies of functional programming learners evaluating recursive functions. *Instr. Sci.* **22**(5), 385–411.

Soloway, E. & Ehrlich, K. (1984) Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.* 595–609.

Soloway, E., Ehrlich, K. & Bonar, J. (1982) Tapping into tacit programming knowledge. Proceedings of the 1982 Conference on Human Factors in Computing Systems. Gaithersburg, USA: ACM, pp. 52–57.

Someren, M. W. (1990) What's wrong? Understanding beginners' problems with Prolog. *Instr. Sci.* **19**(4), 257–282. 10.1007/BF00116441.

Spohrer, J. G. & Soloway, E. (1986) Analyzing the high frequency bugs in novice programs. In Papers Presented at the 1st Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers. Washington, DC, USA: Norwood, NJ, USA: Ablex Publishing Corp, pp. 230–251.

Stefik, A. & Siebert, S. (2013) An empirical investigation into programming language syntax. *ACM Trans. Comput. Educ. (TOCE)* **13**(4), 19.

Thompson, S. & Hill, S. (1995) Functional programming through the curriculum. In *Functional Programming Languages in Education*, FPLE LNCS 1022, Hartel, P. H. & Plasmeijer, M. J. (eds), Nijmegen, The Netherlands: Springer-Verlag, pp. 85–102.

Tinto, V. (1997) Classrooms as communities: Exploring the educational character of student persistence. *J. Higher Educ.* **68**(6), 599–623.

Tirronen, V. & Isomöttönen, V. (In Press) Teaching types with a cognitively effective worked example format. *J. Funct. Program.*

Ulloa, M. (1980) Teaching and learning computer programming: A survey of student problems, teaching methods, and automated instructional tools. *SIGCSE Bull.* **12**(2), 48–64.

Vujoševic-Janicic, M. & Tošic, D. (2008) The role of programming paradigms in the first programming courses. *Teach. Math.* **XI**(2), 63–83.

https://doi.org/10.1017/S0956796815000179 Published online by Cambridge University Press