Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research

Brett A. Becker*
University College Dublin
Dublin, Ireland
brett.becker@ucd.ie

Paul Denny*
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Raymond Pettit*
University of Virginia
Charlottesville, Virginia, USA
raymond.pettit@virginia.edu

Durell Bouchard
Roanoke College
Roanoke, Virgina, USA
bouchard@roanoke.edu

Dennis J. Bouvier
Southern Illinois University Edwardsville
Edwardsville, Illinois, USA
djb@acm.org

Brian Harrington
University of Toronto Scarborough
Scarborough, Ontario, Canada
brian.harrington@utsc.utoronto.ca

Amir Kamil
University of Michigan
Ann Arbor, Michigan, USA
akamil@umich.edu

Amey Karkare
Indian Institute of Technology Kanpur
Kanpur, India
karkare@cse.iitk.ac.in

Chris McDonald
University of Western Australia
Perth, Australia
chris.mcdonald@uwa.edu.au

Peter-Michael Osera
Grinnell College
Grinnell, Iowa, USA
osera@cs.grinnell.edu

Janice L. Pearce
Berea College
Berea, Kentucky, USA
pearcej@berea.edu

James Prather
Abilene Christian University
Abilene, Texas, USA
james.prather@acu.edu

ABSTRACT

Diagnostic messages generated by compilers and interpreters such as syntax error messages have been researched for over half of a century. Unfortunately, these messages which include error, warning, and run-time messages, present substantial difficulty and could be more effective, particularly for novices. Recent years have seen an increased number of papers in the area including studies on the effectiveness of these messages, improving or enhancing them, and their usefulness as a part of programming process data that can be used to predict student performance, track student progress, and tailor learning plans. Despite this increased interest, the long history of literature is quite scattered and has not been brought together in any digestible form.

In order to help the computing education community (and related communities) to further advance work on programming error messages, we present a comprehensive, historical and state-of-the-art report on research in the area. In addition, we synthesise and present the existing evidence for these messages including the difficulties they present and their effectiveness. We finally present a set of guidelines, curated from the literature, classified on the type of evidence supporting each one (historical, anecdotal, and empirical). This work can serve as a starting point for those who wish to conduct research on compiler error messages, runtime errors, and warnings. We also make the bibtex file of our 300+ reference corpus publicly available. Collectively this report and the bibliography will be useful to those who wish to design better messages or those that aim to measure their effectiveness, more effectively.

CCS CONCEPTS

• **General and reference** → **Surveys and overviews**; • **Human-centered computing** → **Human computer interaction (HCI)**; • **Social and professional topics** → **Computer science education**; **Computing education**; *CS1*; • **Applied computing** → **Education**; • **Software and its engineering** → *General programming languages*; *Syntax*; *Compilers*; *Interpreters*; Semantics;

KEYWORDS

compiler error messages; considered harmful; CS1; CS-1; design guidelines; diagnostic error messages; error messages; human computer interaction; HCI; introduction to programming; novice programmers; programming errors; programming error messages; review; run-time errors; survey; syntax errors; warnings

ACM Reference Format:
Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *2019 ITiCSE Working Group Reports (ITiCSE-WGR '19), July 15–17, 2019, Aberdeen, Scotland UK*. ACM, New York, NY, USA, 34 pages. https://doi.org/10.1145/3344429.3372508

*co-leader

ITiCSE-WGR '19, July 15–17, 2019, Aberdeen, Scotland UK

ACM ISBN 978-1-4503-6895-7/19/07…$15.00
https://doi.org/10.1145/3344429.3372508

1 INTRODUCTION & MOTIVATION

Computer programming is an essential skill that all computing students must master [141] and is increasingly important in many diverse disciplines [167] as well as at pre-university levels [166]. Research from Educational Psychology indicates that teaching and learning are subject-specific [135] and that learning programming consists of challenges that are different to reading and writing natural languages [17, 68] or, for example, physics [34]. It is also frequently stated that programming is difficult to learn [127]. As early as 1977 it was explicitly stated that programming could be made easier [189]. The perception that programming is intrinsically hard however, can vary [125]. Regardless, the general consensus for the past several decades from academia to the media, from universities to primary schools, seems to be "Learning to program is hard" [102, p3].

One of the many challenges novice programmers face from the start are notoriously cryptic compiler error messages, and there is published evidence on these difficulties since at least as early as 1965 [177]. Such messages report details on errors relating to programming language syntax and act as the primary source of information to help novices rectify their mistakes. Most programming educators are all too familiar with the reality of messages that are difficult to understand and which can be a source of confusion and discouragement for students. Supporting evidence for this can be found in the literature spanning many decades. Students have reported that such errors are frustrating, and have described them as "barriers to progress" [15]. Further, it has been reported that students have difficulty locating and repairing syntax errors using only the typically terse error messages provided by the average compiler [53, 182]. These are traits not exclusive to programming error messages – general system error messages also cause similar issues [187]. The effects these messages have on students is measurable. A high number of student errors, and in particular a high frequency of repeated errors – when a student makes the same error consecutively – have been shown to be indicators of students who are struggling with learning to program [16, 98]. This area of research has seen increased interest in the last 10 years with authors such as Barik [9], Becker *et al.* [19], Brown & Altadmri [31], Denny *et al.* [50], Karkare (Ahmed *et al.*) [3], Kohn [106], McCall & Kölling [140], Pettit *et al.* [158], and Prather *et al.* [163], focusing on compiler error messages (to name a few).

The feedback that these messages provide are extremely important. A recent survey of hundreds of computing education practitioners revealed that the question "How and when is it best to give students feedback on their code to improve learning?" was rated as one of the most important questions they would like to see researchers investigate [48]. Compilers and programming environments provide feedback immediately, often in the absence of an instructor or lab assistant, and can seem very authoritative, or "all-knowing, infallible authorities about what is right and wrong about code" [113, p109]. Watson, Li & Godwin explained the importance of these messages as follows [207, p228]:

> Feedback is regarded as one of the most important influences on student learning and motivation. But standard compiler feedback is designed for experts - not novice programming students, who can find it difficult to interpret and understand.

Despite this importance, in the last 50+ years, little positive has been said about compiler error messages. In this time they have been described as inadequate and not understandable (Moulton & Miller, 1967) [146], useless (Wexelblat, 1976) [208], not optimal (Litecky, 1976) [120], inadequate [again, 16 years after [146]] (Brown, 1983) [36], frustrating (Flowers *et al.*, 2005) [72], cryptic and confusing (Jadud, 2006) [98], notoriously obscure and terse (Ben-Ari, 2007) [24], undecipherable (Traver, 2010) [202], intimidating (Hartz, 2012) [84], still very obviously less helpful than they could be (McCall & Kölling, 2015) [139], inscrutable (Ko, 2017)[1], frustrating [again, 13 years after [72]] and a barrier to progress (Becker *et al.*, 2018) [21].

Knuth pointed out 'mysterious' error messages as responsible for breaking down the ability to see through layers of abstraction, as Ramshaw, a former graduate student of Knuth's recounted:[2]

> Don [Knuth] claims that one of the skills that you need to be a computer scientist is the ability to work with multiple levels of abstraction simultaneously. When you're working at one level, you try and ignore the details of what's happening at the lower levels. But when you're debugging a computer program and you get some mysterious error message, it could be a failure in any of the levels below you, so you can't afford to be too compartmentalised.

1.1 Motivation

With the users who deal with these messages at the focal point of this work, our chief motivations originate with how programming error messages affect students from two points of view:

(1) **Internal student factors.** For example: Students find programming error messages to be frustrating [21, 51], intimidating [84], as well as confusing and harmful to confidence [100]. Rightfully so, students do not trust them [97] but importantly they do read them [12, 163]. Additionally, their behaviour is altered by them [101] – but most programming instructors have seen that with their own eyes.
(2) **External student factors.** For example: Results which indicate that programming error messages can correlate with traditional measures of success [196, 206], are a measurable part of programmer behaviour [98], result in wasting educator time and resources [200], and the student's experience with programming error messages is highly influenced by the programming environment [18, 101] which plays a vital role when it comes to programming error messages.

We are also motivated by recent findings from Barik *et al.* [12] who have provided empirical evidence that:

(1) Programmers do read error messages (corroborated by [163]);
(2) the difficulty of reading these messages is comparable to the difficulty of reading source code;
(3) difficulty reading error messages significantly predicts participants' task performance, and;

[1] https://blogs.uw.edu/ajko/2014/03/25/programming-languages-are-the-least-usable-but-most-powerful-human-computer-interfaces-ever-invented/
[2] https://www.salon.com/1999/09/16/knuth/

(4) participants allocate a substantial portion of their total task to reading error messages (13-25%).

These authors go on to explain that their results offer empirical justification for the need to improve compiler error messages. In this paper, we argue that these results also highlight a more fundamental truth – programming error messages are a fundamental form of interaction with a system and good error messages are required to make that interaction efficient. In addition we have little at our disposal to actually measure how efficient that interaction is.

When considering novices, perhaps the stakes are highest. Twenty years ago, Kölling stated [107, pp145-146]:

> Good error messages make a big difference in the usability of a system for beginners. Often the wording of a message alone can make all the difference between a student being unable to solve a problem without help from someone else and a student being able to quickly understand and remove a small error.

In 2016 (17 years later) McCall & Kölling noted that Java error messages in particular, are not only (still) confusing from the novices' point of view, but probably confusing for all programmers due to the following observations [138, p2]:

- A single error may, in different context, produce different diagnostic messages.
- The same diagnostic message may be produced by entirely different and distinct errors.

Finally, in 2019 the same authors reiterated that the information provided by (some/many) compiler error messages is inaccurate and imprecise: "This information, however, does not have a direct correlation to the types of errors students make, due to the inaccuracy and imprecision of diagnostic messages" [140, p38:1]. They also provide a rich discussion on the issues with error *messages*. The quote below is only a sample of a much larger discussion in [140, p38:20]:

> A compiler can produce two different diagnostic messages for what is logically the same error occurring in two subtly different syntactical contexts, for example; similarly, it might produce the same diagnostic message for two quite distinct errors that could be recognised by an experienced programmer as logically distinct error types. Furthermore, different compilers – or different versions of the same compiler – may produce different diagnostics when presented with the exact same source code input.

If inconsistency within a specific compiler is an issue, it complicates matters further when the differences between compilers are considered. Different compilers producing different errors for the same code has been (sporadically) studied for nearly 40 years [27, 35].

The authors of this report have first-hand experience with students struggling with programming error messages, like many other programming instructors. One author had a student remark the following recently: "If I had to pin-point the most difficult aspect of C so far I'd have to say making sense of error messages". Another author had a student comment (referring to Java) [14, p1]: "If a compiler error was worth one Euro, I would be a millionaire".

Marceau, Fisler & Krishnamurthi nicely summed up the situation [133, p3]:

> Yet, ask any experienced programmer about the quality of error messages in their programming environments, and you will often get an embarrassed laugh. In every environment, a mature programmer can usually point to at least a handful of favourite bad error responses. When they find out that the same environment is being used by novices, their laugh often hardens.

Although we focus on student programmers it is important to realise that errors also affect professional programmers in non-trivial ways. Studying 26.6 million builds generating 57.8 million error messages at Google, Seo *et al.* found between 28% and 38% build failure ratios depending on language, that 10% of error types account for 90% of build failures, and that the resolution time was highly dependent on the error message, ranging from around a few minutes up to an hour with a median of about 5 minutes for C and 12 minutes for Java [183]. Rough calculations reveal that the median time spent on failed builds is 4.7 hours per month for C++ and 5.7 hours per month for Java. Traver [202] also pointed out that experts have interests in error messages, noting that when changing languages they are in some ways beginners again [181], and that improved messages benefit professionals [116].

1.1.1 A Motivating Example. As a motivating example, consider the following Java code which is typical of a first programming example or assignment in a first-year university introductory programming course.

```
1 public class hello {
2     public static void main(string[] args) {
3         system.out.println("Hello World!");
4     }
5 }
```

The two error messages resulting from attempting to compile the above code are shown below.

```
hello.java:2: error: cannot find symbol
  public static void main(string[] args) {
                              ^
  symbol:   class string
  location: class hello

hello.java:3: error: package system does not exist
  system.out.println("Hello World!");
  ^
2 errors
Process Terminated ... there were problems.
```

Even in this simple example, neither error message accurately reflects the true programmer-introduced errors that generate them. They may accurately reflect what the compiler thinks is wrong, but this is likely meaningless to a beginner. Simpler descriptions of the

above errors would have a greater chance of being understood by a beginning programmer. For example:

- The “s” in “string” on line 2 should be capitalised.
- The “s” in “system” on line 3 should be capitalised.

Of course, this is easier said than done. In general the compiler cannot divine the intention of the programmer. Making error messages more specific – as illustrated by our suggestions above – runs the risk of proposing solutions that the user does not intend.

Finally, if cryptic messages are confusing for fluent speakers of English (the language upon which most programming languages and their documentation are based), this situation may present additional barriers to non-fluent speakers of English [17].

1.2 Outline

This paper is laid out as follows. In Section 2 we provide a brief background to this work, describe our objectives and audience, and formalise our nomenclature. In Section 3 we provide precise definitions for the terms used in the paper and review how common program analysis tools generate error messages. We conduct a comprehensive review of the literature and report our methods and findings in Section 4. In the following four sections, we examine the literature in greater detail around several prominent themes. Firstly, in Section 5, we report findings from a pedagogical perspective. In Section 6, we explore how researchers and tool designers have addressed the technical challenges of generating effective error messages. We then review attempts to improve, or enhance, error messages themselves, summarising both historical and recent efforts in Section 7. Finally, in Section 8, we organise guidelines curated from the literature – suggested by researchers over nearly six decades – for designing useful error messages, and present evidence supporting (and not supporting) them. We conclude this paper with a summary of our main findings in Section 9.

2 BACKGROUND & APPROACH

Diagnostic error messages generated by compilers and interpreters, including error, warning, and run-time messages, have been researched for over 50 years with one obvious consensus: they present substantial difficulty and could be more effective [18, 163]. They are often vague, imprecise, confusing and at times seemingly incorrect [36], in particular for novices. Research has also shown that these messages can be a source of frustration for students [21, 51]. Unfortunately, drawing any more specific conclusions from this history of research is difficult. In 1983, Brown stated “There should be a deliberate and sustained effort to focus attention on the quality of error messages, both in compilers and other systems, so that the current appalling state-of-the-art can be improved” [36, p249]. In 1984, du Boulay & Matthew asked “Why cannot these messages be made clearer, less confusing and more accurate?” [57, p109]. We argue that du Boulay & Matthew’s question is still valid, and Brown’s call for action has not been met. We also note that educators strive for their feedback to students to be accurate [126], and we should expect the same from the tools we have students use.

Although in recent years there has been increasing interest in programming errors (faults committed by students) and the diagnostic messages they generate [22, 127], the literature on these messages is quite scattered and has not been brought together in an easily digestible form. This is despite the fact that the presentation of empirical evidence on the effects of these messages has advanced recently [192]. Various studies have analysed the types and frequency of diagnostic messages that students generate; others have explored how ‘standard’ messages can be enhanced to make them more usable; and others still have sought to determine how the effectiveness of these messages can best be measured. Yet the difficulties these messages present, and evidence for their effectiveness, have not been comprehensively analysed.

This is exacerbated by the fact that although some works have presented evidence on diagnostic message effectiveness [15] (or lack thereof [50]), this evidence seems to be conflicting [158], but may not be as conflicting as it appears. Additionally, some message design guidelines (explicit and implicit) exist but they span several decades and many of them are also conflicting, leaving the way forward unclear. We argue that to help the community proceed with more work on such messages, a clear picture of the state-of-the-art in compiler/interpreter diagnostic message design and effectiveness is required. In addition, a synthesis of the evidence on these messages including the difficulties they present and their usefulness would be helpful. A literature- and evidence-backed set of guidelines for the design of diagnostic programming messages would also be a valuable resource for the community. That is what this report strives to deliver.

It is worth noting that the lessons learned in this arena may be generalisable across languages, cohorts, and tools. Motivated by writing error explanations for novices, Pritchard [164] found that in large datasets of Java and Python programming error messages, the frequencies of message (error) types empirically resemble Zipf-Mandlebrot distributions. One possible implication of this work (beyond an interesting and possibly useful way to compare languages) is in measuring what languages give more distinctive programming error messages. Becker found very strong similarities in the distributions of the ten most frequent Java messages from six studies spanning several years and Java versions [14] (see Figure 1).

Jadud [98] found a less common but still similar distribution between six languages/environments: BlueJ (a pedagogic Java environment [109]); COBOL [120]; Helium (a pedagogic Haskell Environment [87]); LOGO [157]; and SOLO (a LOGO-like language for manipulating semantic networks [61]).

On the messages themselves, Traver gives a relatively current and quite comprehensive treatment [202]. Barik’s works also provide quite current insights coupled with empirical findings and justifications for programming error messages to be investigated with more seriousness than in the past [9, 10].

2.1 Audience & Objectives

The intended audience of this paper falls into four broad and often overlapping categories:

(1) language designers
(2) tool developers
(3) educators
(4) researchers

Although these categories can overlap significantly, in this paper we provide directed discussions aimed at one or more of these groups. We note that this categorisation also includes novice programmers

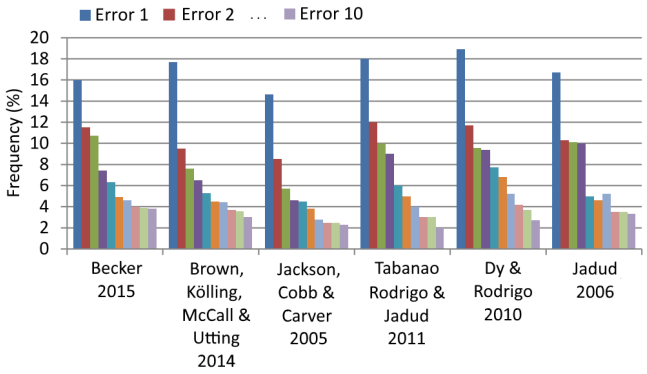

Figure 1: Frequency of the ten most common Java errors from six Java studies: Becker [14]; Brown *et al.* [33]; Jackson *et al.* [96]; Tabanao *et al.* [196]; Dy & Rodrigo [59]; and Jadud [98]. Image adapted with permission ©Brett A. Becker [14].

indirectly, through these four audiences, as there are designers, developers, educators and researchers that all interact with novices often and on different levels. Despite this, we also believe that this paper will be helpful for those who are interested in how programming error messages affect more experienced students and even professional programmers. We intend this report to be useful for the above categories of audience not just in Computing Education, but in several different sub-disciplines such as Human Computer Interaction, Programming Languages, and Software Engineering. We hope that our efforts can contribute to more progress in the research of diagnostic programming messages on many fronts.

We aim to satisfy two broad objectives with this work. The first of these is to present a state-of-the-art report encompassing the difficulties that diagnostic messages present (particularly for novices), the design of these messages, and evidence of their effectiveness. This brings together the literature and evidence on diagnostic programming messages from the disciplines of Computing Education, Human Computer Interaction, Programming Languages, and more. The second objective is to present a unified set of guidelines for the design (or improvement) of these messages, which we hope will inform the future work of language designers and tool developers. Some studies over the last few decades have provided concrete design guidelines — for instance, see [202]. However, not all are backed by empirical evidence, and some are conflicting [158], which highlights the necessity for further research. Based on what is currently reported in the literature, we present a unified set of guidelines that can be used as a starting point for designing or improving diagnostic messages.

2.2 The Nomenclature of Errors and Messages

The nomenclature of programming errors and messages differs from community to community and even researcher to researcher. In this paper we have already used the terms: 'error message', 'diagnostic error message', 'syntax error message', and 'compiler error message'. In part this is deliberate - all of these terms have been used many times, for many years, by many people, to refer to the same thing. However we do not choose to continue using the first two as they are too vague. We choose not to use 'syntax' or 'compiler', despite widespread usage, as these may seem to exclude, for instance, runtime error messages and non-compiled languages.

After much discussion, including with those beyond the authors of this report, we have decided to use the term *'programming error message'* in the remainder of this report. We are aware that 'error' could seem to exclude warnings, for instance. Nonetheless we find that excluding error completely loses meaning. After all, most of the messages we are discussing are due to errors. In turn these errors are committed by the programmer. Marrying 'programming' with the 'error' and the 'message' we feel completes the feedback loop that occurs when programming: the programmer writes a program as input and errors cause the machine to return messages back to the programmer. We view this as an analogue to Jadud's *edit, compile, run* cycle [98], but from a feedback/message point of view. We note that some authors prefer 'diagnostic message' (specifically over "compiler error" [140]). However after much discussion (again, including with many outside of the authors of this paper) and very closely favouring 'diagnostic' we concluded that 'diagnostic message' or even 'diagnostic error message' is too vague. In the case of the latter for instance, it is clear that the message is intended to diagnose an error, but this could for instance be referring to a system error, an HTTP 404 error, or any other error. When absolutely necessary we use terms such as: runtime error message, warning message, etc. when we need to discuss a specific type/category of message.

However, what constitutes a message? It turns out that this is also a possibly contentious word. Later in Sections 3.2 and 3.3 we precisely and technically define the words *error* and *message* for the purposes of this paper. However at this point it is appropriate to remind the reader that this work focuses on *messages* – text returned to the user from the programming environment due to an *error in code* or other violation of language specification *committed by the programmer*. As programmers often commit errors which often generate messages, we necessarily discuss errors themselves, on occasion. However, there is a vast literature on *errors* which largely is beyond the scope of this paper. As an entry point into the literature on errors themselves, as well as many other facets of novice programmer behaviour, we recommend [127].

3 PRELIMINARY DEFINITIONS

We frame our discussion of programming error messages by first precisely defining our objects of study. Commonly used terminology regarding programming languages is frequently tied to the particulars of a language and its implementation. Sometimes, this terminology is even contradictory between languages; for example, syntax errors in Python are considered static errors (*i.e.*, before interpretation) whereas they are considered dynamic errors in JavaScript (*i.e.*, during interpretation). Since our work is programming language agnostic, we strive to define programming errors and the surrounding terminology in a similar way. To do this, we first review basic concepts and terminology about compilers, interpreters, and programming languages. Programming errors are intrinsically tied to the compilation process, so it is important to first review the architecture of compilers and interpreters. With these definitions

in hand, we can arrive at a general categorisation of programming errors that will guide the rest of our discussion.

3.1 The Architecture of a Compiler

Compilers and interpreters are two instances of *program analysis tools*. Other sorts of program analysis tools include linters, debuggers, code editor assistants, and visualisers. Regardless of their functionality, all program analysis tools perform two fundamental operations: *translation* of a program from one representation to another and *analysis* of that program in a particular representation. Translation allows a tool to improve code in some way, *e.g.*, through formatting or optimisation, or transforming the code into a form that is ideal for a particular analysis. These analyses gather information about the program for use by the developer, *e.g.*, auto-completion tools, or for internal use, *e.g.*, to ascertain the correctness of the program.

Due to memory and processing constraints, classical tools were built as monolithic structures, intertwining these two operations in ad hoc ways. Modern program analysis tools, in contrast, are structured as a *pipeline of passes*. Each pass is responsible for taking a program representation as input and producing a new representation as output, performing some amount of analysis in the process.

There are several passes that are common among many program analysis tools:

- *Lexical analysis* transforms raw source code (a string of characters) into a sequence of *tokens*, chunks of meaningful text, *e.g.*, identifiers and punctuation.
- *Parsing* transforms a sequence of tokens into an *abstract syntax tree* (AST), a tree-like representation of the program amenable to analysis.
- *Static type checking* analyses an AST to ensure a program is consistent with the language's type system (if that language is statically-typed).
- *Lowering* transforms an AST into a lower-level representation amenable to further analysis or direct execution by an interpreter or the machine itself.
- *Interpretation* directly executes a program, usually in its AST representation or some lower-level form appropriate for efficient execution.

Other passes include pre-processing passes (source-to-source translations), optimisation passes (source-to-source translations with the goal of creating a more efficient, yet equivalent program), and linking passes (lower-level translations that combine collections of compilation units into a single representation).

Static versus Dynamic Analyses. A common distinction to make is whether a language analysis occurs during program execution. We use the term *dynamic* or *runtime* to refer to any action that occurs during program execution and the term *static* or *compile-time* to refer to an action that does not occur during program execution. This is critical with respect to compiler errors since more information about the state of a program is available dynamically, generally allowing for greater precision in error messages. However, it is also often preferable to report errors as early as possible, making static detection attractive, and leading to a fundamental trade-off between precision and locality when designing error-handling systems for these tools.

Compilation and Interpretation. Compilers and interpreters are the primary program analysis tools used by developers. At a high level, *compilers* translate textual source code to a machine-executable form, and an *interpreter* directly executes textual source code. However, our understanding of program analysis as a pipeline of passes helps us better understand how the two sorts of tools are closely related.

Both compilers and interpreters perform a series of successive translations and analyses of source code. Virtually all tools use a lexing and parsing pass to transform the initial source into an abstract representation, usually an abstract syntax tree (AST). These tools will also perform different sorts of static analysis on the AST, notably type checking for statically-typed languages. From here, these tools will continue translating the program to different representations and perform additional analyses until the program reaches a final form, *e.g.*, LLVM IR, Java bytecodes, assembly, or even the original AST if no additional translation was performed. This is the point at which compilation and interpretation differ: a compiler produces the final form of the program as output, and an interpreter executes the final form of the program.

This fluidity is best demonstrated when traditional 'compilation' and 'interpretation' tasks happen dynamically and statically, respectively. The first case occurs with *Just-in-time compilers* (JITs) commonly found in languages such as Java and C#. In the case of Java, the Java compiler (`javac`) first translates a program into Java bytecodes (`.class` files), an intermediate format amenable to interpretation by the Java Virtual Machine (JVM). The JVM interprets these bytecodes directly, but during interpretation, it also translates bytecodes that are frequently run directly to machine code, *e.g.*, x86 assembly, as a performance optimisation. The second case occurs with *macro systems* such as those found in the Racket programming language[3]. In Racket, macros are simply Racket programs that produce new Racket source code as output and are interpreted at compile-time in a pre-processing pass.

3.2 Programming Errors

With a better understanding of how compilers and interpreters are structured, we can precisely define programming errors. There are two broad ways in which a program may be incorrect, implying that it contains a *programming error*:

(1) The program is not well-formed according to the specification of the language.
(2) The program is well-formed but does not behave according to its own specification of correctness.

We say an error is *caught* (and thus an appropriate error message can be produced) whenever a system, whether it is an analysis tool like a compiler or the program itself, detects and intentionally responds to an erroneous program state with a message. Errors can be caught either statically or dynamically, depending on how a particular language is implemented. Note that this stands in contrast to *user errors* that arise due to the *user* breaking their contract with the program, *e.g.*, by providing erroneous input, rather than the programmer doing something wrong. Such errors are out of the scope of this discussion.

[3] https://racket-lang.org/

When an error is not caught, we say that the error is *silent*, leading to *undefined behaviour* at run time. Undefined behaviour can include data corruption, arbitrary program behaviour, simply returning incorrect results, or crashing without a message reporting the error at all.

Language Specification Errors. Programming errors that are detected by a compiler are categorised by the phase of the compilation process that detects the errors.[4] A programming language defines the requirements of a well-formed program through its *language specification*: what is *syntactically* and *semantically* acceptable. Syntax concerns the grammatical structure of a program – are the different components of a program arranged correctly according to the language's grammar? *Syntax errors* arise during the lexing and parsing passes of analysis when a program is checked to ensure that it is syntactically correct. In contrast, semantics concerns the meaning of a program—can it be run to produce a meaningful result? The different *semantic errors* are further categorised based on the pass in which they are caught. Most notably, *type errors* are caught during type checking, whether it occurs statically during a type-checking pass or dynamically during program execution. Other sorts of semantic errors are possible depending on the kinds of analyses a particular tool performs and the behaviour of the program, *e.g.*, *scoping errors* when variable use is not in scope of its definition, *memory errors* when memory safety is violated, and *concurrency errors* when multi-threaded code is not synchronised appropriately.

Program Specification Errors. On top of the language specification, each program has its own program-specific specification. This is not necessarily a formal specification; it is merely what the developer intends their program to do. A program analysis tool does not know this specification *a priori*, and so, by definition, it cannot detect when a program violates its own specification. If the developer cares to validate this specification, a process called *program verification*, they must do one of two things:

(1) *Externally* validate the program against this specification in some way, *e.g.*, through testing or formal proof.
(2) If the language allows it, make parts or the entirety of the specification known to the program analysis tool, *e.g.*, through types or assertion statements, so that the tool can validate the specification *internally*.

By taking the latter approach, the programmer turns would-be program specification errors into language-specification errors by tying the two together.

Program verification is a widely active area of research within computer science and encompasses a broad range of tools, both internal and external to the compiler. We limit the scope of our discussion to verification tools used in mainstream compilers and interpreters, primarily type systems. However, we hope the guidance in this report can help inform programming error message design for all of these verification tools.

[4]This is really a language implementer's perspective on programming error categorisation. It is important to realise that other sorts of categorisations are possible, *e.g.*, based on the kinds of conceptual misunderstandings that lead to an error [139], which can cause confusion when cross-disciplinary peers discuss this subject matter.

3.3 Programming Error Messages

In this work, we focus on text-based programming error messages typical of those returned to the user when compiling at a command-line. Many IDEs also return these messages to the user through an interface. Sometimes these environments alter these text-based messages. For instance, BlueJ (up to version 3.x) returned only the first message to the user, even if compiling the same program at the command-line (using the same javac version) would return multiple messages as a result of a single compile [21]. Nonetheless, the IDE is still presenting text-based messages to the user. However there are other ways of identifying errors which we do not consider to be messages. Examples include red-squiggle underlining used by many environments such as Eclipse and BlueJ 4. We also exclude icon-based notifications such as those in NetBeans. Often however, clicking on or hovering above underlined code or these icons does reveal text-based messages (normally in a pop-up).

4 CORPORA

Our examination of the existing programming error message literature utilised two separate sources. One source was the result of a quasi-systematic search, using a validated search string, over several relevant electronic databases. This search was conducted on 15th June, 2019. The other source was a large body of literature collected in an ad hoc fashion requiring, at times, much manual effort. Most of this literature was collected over several years by the first author of this report, however it remains a work in progress and is constantly being updated as new work is published (and old work is unearthed). The numbers we present in this section reflect the state of this manually curated collection as of 21st August, 2019.

One of the contributions of this research is a comprehensive corpus of the literature on programming error messages which we formed by combining these two sources. Currently, our corpus consists of 307 articles – a bibtex formatted listing of this literature is available online.[5] In the next two sections we describe each of our two sources – the manually curated corpus and the result of the quasi-systematic search – and present the methods used to generate, analyse and merge them.

4.1 Original Corpus

The process of collecting the programming error message literature, as used in this research, began in 2012 by the first author of the paper. Like many other teachers, this author witnessed students struggling with error messages and decided to write a Java editor (Decaf) that intercepted and enhanced the standard javac error messages [15, 18]. At this point, the author started collecting references on the literature surrounding error messages. This process picked up pace when studying the effects of Decaf became the topic of a thesis [14]. Later, two PhD students of the first author, both doing work in similar areas, helped contribute to the corpus. Many of the items in the corpus are not available online (but may be available from the authors[5]) and several were difficult to find, including:

(1) several theses that are not available online, some obtained from the authors or libraries (some on microfilm)

[5]https://iticse19-wg10.github.io/

(2) sources that are out-of-print and not available online obtained through inter-library loan
(3) personal correspondence with authors
(4) unpublished works
(5) magazine-style publications

Although referred to as the 'original' corpus, this collection of articles is a work in progress and is updated frequently as new relevant literature is published. As of 21st August, 2019, this corpus consisted of 192 articles. One unique aspect of the corpus is the number of theses that focus on (or are very related to) the enhancement of programming error messages, including [9, 14, 41, 80, 81, 84, 89, 98, 99, 105, 107, 119, 138, 143, 200, 218].

4.2 Quasi-systematic Search

The corpus of literature described in Section 4.1 provided a solid starting point for this research. We were, however, conscious that the methods used to collect this literature were not systematic or repeatable. As a result, an implicit bias reflecting the specific interests of the author may have impacted the coverage of the corpus, and there may be literature relevant to programming error messages absent from the corpus.

To address this, we conducted a quasi-systematic review of the literature on programming error messages, informed by the guidelines proposed for such reviews by Kitchenham & Charters [103]. These guidelines were also cited by recent large-scale reviews of the literature on introductory programming [127, 144].

One of the commonly stated reasons for conducting systematic reviews is to provide a complete background on which to position new research activities [103]. This aligns with our goals, as we hope our paper will assist the community in proceeding with new work on programming error messages.

4.2.1 Research Questions. We were aware, from reading through the articles in our original corpus, that programming error messages are reported in the literature in a wide variety of ways. Examples include descriptions of the difficulties that students encounter with error messages, suggestions for improving the wording of error messages and empirical studies measuring their impact, and reports of the relationship between errors and error messages encountered by students with their performance. Our goal with the search was to catalogue this wide-ranging literature on programming error messages. As such, our overall research question was broad:

- In what ways have programming error messages been reported in the literature?

We were particularly interested in how error messages can be improved, leading to specific research questions focusing on technical challenges, published guidelines and empirical studies on error message enhancement:

- What are the technical challenges around providing good error messages?
- What guidelines have been reported for constructing effectively worded error messages?
- What evidence exists that the wording of error messages has an impact on student programming behaviour and learning?

4.2.2 Search String and Validation. Our search process began with manual inspection of the proceedings of two recent conferences. The goal of this manual inspection was to assist with subsequent validation of the search string used for the quasi-systematic search. Once the manual search identified all relevant papers in a given proceedings, the search string could be applied and limited to the corresponding conference to reveal false positives (papers returned by the search string but deemed not relevant) and false negatives (papers identified as being relevant but not returned by the search string).

For this manual stage, we selected the SIGCSE 2018 and 2019 conference proceedings. These were selected as we were aware they included several papers related to our topic. In the first step of the process, we read the titles and abstracts of all 330 papers of four or more pages in both proceedings. This resulted in 52 potentially relevant papers for which we examined the full paper, in total identifying 5 papers from these two conferences that focused on programming error messages.

We experimented with numerous search strings across a range of electronic databases. Our experience working with search strings and the different indexes mirrors that of Brereton *et al.*, who report a number of challenges given the different models around which the databases are organised [30]. Luxton-Reilly & Simon *et al.* also specifically noted the trade-off we experienced between the use of more general search terms and generating a manageable number of results [127]. We discovered one more issue that requires care in reporting the number of matching articles – for some indexes, the number of results shown on the webpage differs from the number of articles that are actually downloaded through the database's exporting feature (some of this discrepancy appears due to duplicate articles, but with unique DOIs, appearing in the exported list). We therefore report both numbers below.

We use the following search string:

> "error message" AND ("compiler" OR "diagnostic" OR "interpreter" OR "programmer" OR "programming" OR "syntax" OR "semantic")

We recorded results from four indexes: the ACM Digital Library, Scopus, Science Direct and IEEE Xplore. To validate the search string, we applied it to the ACM Digital library and limited the search to the SIGCSE 2018 and 2019 proceedings. This returned exactly the 5 papers we had previously identified manually (in addition to one abstract-only article which we exclude from consideration due to its length).

We conducted the full search on 15th June, 2019, using metadata (title, abstract and keyword) searches on the following four databases with the following results:

- ACM DL Full Text Collection: 154 reported, 189 exported
- Scopus: 377 reported and exported
- Science Direct: 72 reported and exported
- IEEE Xplore: 18 reported, 13 exported

4.2.3 Selection and Filtering. A total of 651 articles were returned from the combined searches. After removing duplicates, and excluding articles of fewer than four pages, a total of 448 articles remained for manual inspection. These were assessed for relevance, and excluded if they were written in a language other than English, less than four pages in length, not clearly about programming error messages or not relevant for answering our research questions. Of

the 448 articles examined, 285 were excluded, leaving a total of 163 articles for classification.

4.3 Classification

As previously described, the working group identified literature on programming error messages though two methods:

(1) the original corpus of 192 articles identified manually over years of working on many distinct yet related efforts, and;
(2) the 163 articles identified through the quasi-systematic search described in Section 4.2.

A total of 48 articles appeared in both the original corpus and the quasi-systematic search results. In other words, approximately 70% of the quasi-systematic search results represented new articles not uncovered by our manual search efforts. Initially we were surprised at the relatively small overlap between our two sources, however we believe this reflects the very diverse nature of the existing literature on programming error messages.

To effectively utilise this corpus, a tagging system was proposed. All 12 authors collaborated on the development of a set of 13 tags and accompanying descriptor statements as shown in Table 1. These tags were developed and refined through several weeks of discussion and testing. Once agreed, the papers were tagged by the authors in an on-going, iterative, constantly evolving process. This began with each author tagging a unique subset of the papers, after which any author, at any time, could tag, re-tag or un-tag any paper. This was achieved by entering all references into the Mendeley reference management software and tagging according to the tag descriptors in Table 1. In Table 1, rows are ordered from most to least frequently used tags overall. Each paper may have had more than one tag assigned (theoretically up to 13), so the column totals exceed the number of papers. The average number of tags per paper was 2.1 (min = 1, max = 8, standard deviation = 1.4).

This corpus of unique articles on programming error messages consists of 307 papers. The bibtex entries for these papers are available online.[6] Figure 2 illustrates the distribution of articles across publication years, broken down by source. In recent years, there has been a sharp increase in the number of published articles relating to programming error messages, perhaps reflecting the growing number of tools and languages in popular use. Well over half of the papers in the corpus were published in 2011 or later. There doesn't appear to be any obvious bias in the years over which our quasi-systematic search uncovered literature compared with our manually curated original corpus.

Several major themes emerged from the classification of papers in the corpus. The 'enhancement' tag was the most commonly used, with 107 papers identified as proposing or studying modifications or enhancements to programming error messages for an existing system or language. Many of these articles focused on the technical challenges of generating useful error messages, in some cases describing very low-level details, and were also tagged as 'technical'. The distinction between 'guidelines' and 'pre-guidelines' was sometimes subtle, with the former tag reserved for articles that provided concrete lists of guidelines for designers of error messages. In the following subsections we expand on each of these four main themes. In Sections 5, 6, 7, and 8 we discuss these in detail.

[6] https://iticse19-wg10.github.io/

4.3.1 Pedagogy. How programming error messages affect students, their learning, and the teaching of programming are central to our study of these messages and for that reason we will address this tag first. Perhaps somewhat unsurprisingly (to the authors at least) the pedagogy tag was mid-table in terms of representation. In our combined corpus, a total of 43 papers (14% of the entire combined corpus) were tagged as 'pedagogy'. Of these papers, 37 were collected manually as part of our 'original' corpus while our quasi-systematic search uncovered 19 (13 of these appeared in both sources). In Section 5 we discuss the main findings on this front.

4.3.2 Technical. Somewhat unsurprisingly, the 'technical' tag was second most frequent. The key technical challenges to improving the handling of programming errors are error detection, localisation and providing feedback. Any useful system must detect any deviation from the specification of a programming language (error detection). It must accurately report the locations where such errors have occurred (localisation) and finally, it must provide a message to help users understand and fix the errors. Understanding and overcoming these technical challenges is often the first step in improving the quality of generated error messages, and we find many of the papers in our corpus are tagged as "technical". An examination of the author-defined keywords for these articles, extracted from our corpus, reveals most describe techniques based on type systems and type inference. Other keywords include compilers and parsers, static analysis, dynamic analysis and run-time error messages.

In our combined corpus, a total of 91 papers (30% of the entire combined corpus) were tagged as 'technical'. Of these papers, 35 were collected manually as part of our 'original' corpus while our quasi-systematic search uncovered 72 (16 of these appeared in both sources). Section 6 elaborates on the technical issues related to programming error messages in detail, a topic not often discussed in computing education venues.

4.3.3 Enhancement. Given the struggles that programmers, especially novice programmers, have with error messages, improving or enhancing the messages is of great interest to both the programming languages and education communities. In our combined corpus, a total of 107 papers (35% of the entire combined corpus) relate to the enhancement or improvement of programming error messages and were tagged as 'enhancement'. Of these papers, 72 were collected manually as part of our 'original' corpus while our quasi-systematic search uncovered 64 (29 of these appeared in both sources). In determining exactly what constitutes an 'enhancement', for the purposes of our working group, we considered papers that replaced existing error messages generated from the system, altered these messages in any way, or added tips or extra information to the messages. We did not include, for instance, efforts to identify syntax mistakes dynamically within an IDE by indicators such as red 'squiggly' lines. Section 7 discusses attempts at, and results from, error message enhancement in more detail.

4.3.4 Guidelines. Papers that presented guidelines for designing useful messages were quite common in our corpus. We classified such papers as either 'guidelines' or 'pre-guidelines' with the former assigned to 35 papers that offered explicit rules or clear concrete guidelines and the latter assigned to 50 papers that offered only

Table 1: Frequency of tags assigned to papers from the original corpus, the quasi-systematic search results, and papers that appeared in both (overlap). Note that multiple tags could be assigned to each paper.

Tag	Descriptor *papers that …*	Original and quasi-systematic	Original only	Quasi-systematic only	Total
enhancement	…propose modifying or enhancing existing error messages	29	43	35	107
technical	…describe technical issues or challenges with providing better error messages	16	19	56	91
empirical	…present empirical results	28	44	17	89
difficulties	…describe the difficulties that programmers have with error messages	24	44	10	78
justification	…provide a strong justification for studying programming error messages	14	38	5	57
pre-guidelines	…offer suggestions for forming guidelines, but which do not provide concrete guidelines	16	25	9	50
pedagogy	…discuss the relationship between messages and pedagogy	13	24	6	43
guidelines	…include explicit guidelines, rules, or concrete suggestions for designing error messages	11	16	8	35
errors-only	…discuss errors that specifically do not mention diagnostic messages	1	16	8	25
anecdotal	…provide anecdotal results relating to programming error messages	5	17	1	23
tool	…describe a tool that uses error messages, but doesn't create or enhance them	6	4	10	20
performance	…establish a link between errors/messages and programmer performance	5	6	4	15
runtime errors	…discuss runtime error messages	2	2	4	8

general suggestions. Of the 'guidelines' papers, 27 were collected manually as part of our 'original' corpus while our quasi-systematic search uncovered 19 (11 of these appeared in both sources). For papers tagged 'pre-guidelines', 41 were collected manually, and 25 resulted from our quasi-systematic search (with an overlap of 16 papers).

The literature reports a wide range of guidelines and suggestions for generating useful error messages. These are scattered across venue, time, and communities. We found four notable sources that contained a wealth of guidelines, dating from 1976 to 2018: Horning (1976) [90], Shneiderman (1982) [187], Traver (2010) [202], and Barik (2018) [9]. We present these guidelines side-by-side, along with guidelines from other authors in Section 8.

4.3.5 Summary. As we combined the 'pre-guidelines' and 'guidelines' tags for the purpose of discussion in this paper, the four subsections above represent five of the top eight tags. We do not directly discuss 'empirical', 'difficulties', or 'justification' as most of these papers support the tags we do discuss. To a lesser extent the same is true for 'anecdotal' which was the 10th most frequent tag. Additionally, we do not directly discuss the five least frequent tags. Excluding 'anecdotal', the most frequent of the bottom four tags is 'errors-only', which we don't directly discuss as described in Section 2.2. The remaining three tags are ('tool', 'performance', and 'runtime errors'). A manual inspection revealed that only three papers with any of these three tags did not have at least one other tag from the remaining 10 tags. Further inspection reveals that two of these three papers are cited in this paper. By inspecting the final remaining paper, we can reasonably determine that although the current paper does not cite all 307 papers in our corpus, each paper was considered for discussion, not only tagging.

5 PEDAGOGY & EDUCATIONAL CONTEXT

The detrimental effects of cryptic error messages on novice programmers have appeared throughout the literature for decades [18, 77, 98, 110, 111, 113, 138, 208]. Educators are affected indirectly as they must devote time to helping students correct programming errors when the messages cannot be understood [14, 41, 72, 193]. Often this involves explaining the same error message to multiple students repeatedly.

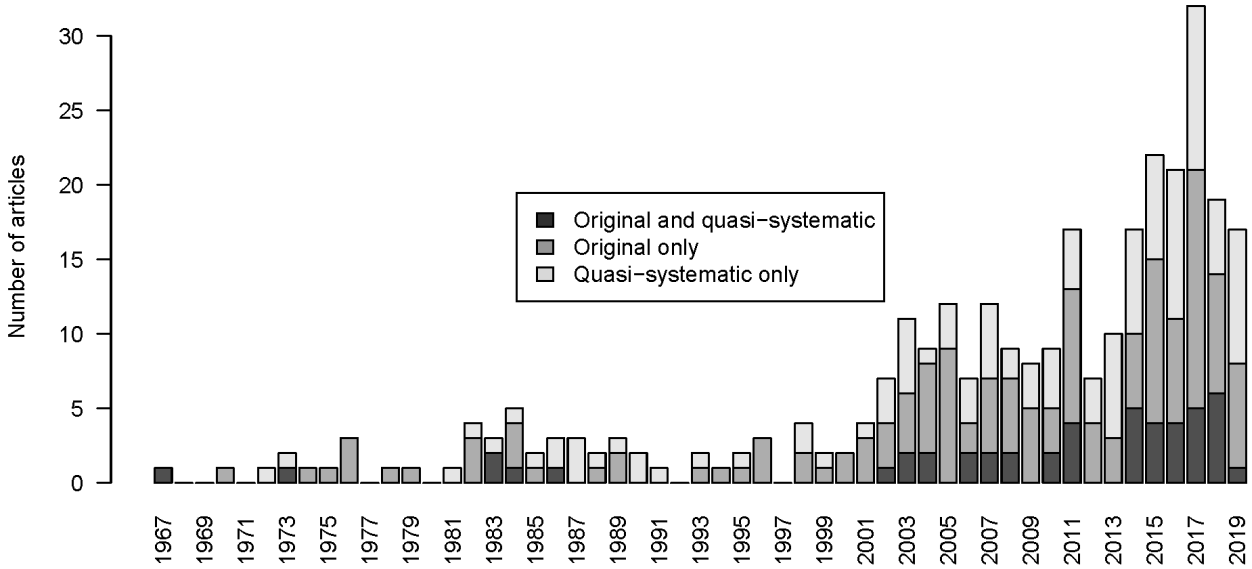

Figure 2: Publication years for articles appearing in our final corpus, broken down by source. The darkest 'Original and quasi-systematic' represents articles that appeared in both the original corpus and our search.

Educators designing programming languages with novice programmers and other learners in mind have all had to grapple with the presentation of errors, so it is not surprising that the topic of how error messages affect the learning of programming has also been discussed by the educational community since at least 1965 – the date of the earliest publication we found, which specifically discussed FORTRAN diagnostic messages and students [177].

Although not designed as a teaching language, FORTRAN was used for teaching, especially in the early years of computing education. FORTRAN's programming error messages were deemed so difficult for learners that at least four implementations of FORTRAN were devised that each improved on standard FORTRAN messages, all developed with students in mind. This is fairly remarkable given the much lower number of publications from this era, particularly in computing education. DITRAN (*DI*agnostic FOR*TRAN*) [146], WATFOR (*WAT*erloo *FOR*TRAN) and WATFIV (WATFOR's successor) [43], and PUFFT (Purdue University Fast FORTRAN Translator) [177, p661] are discussed in more detail in Section 7.

Logo, BASIC, and Pascal were all significant and highly utilised languages in the earliest era of programming languages designed specifically for learners. Logo was designed in 1967 as an educational programming language with design principles grounded in Piaget's research into how children develop thinking skills. Notably, it was designed with the inherent goal of having informative error messages. Harvey describes this intention as, "a language for learners has to be designed to deal with problems that are less important in a language meant for experienced programmers. For example, when you make a mistake, you should get a detailed, helpful error message" [85, p33]. Attempting to address some similar issues, one of the original requirements of the BASIC language, which according to Kurtz, was developed for liberal-arts students, was that "the system would be pleasant and friendly" [111, p106]. In reflecting back on the development of BASIC, Kurtz said, "BASIC also shows that simple line-oriented languages allow compact and fast compilers and interpreters, and that error messages can be made understandable" [111, p117]. Following BASIC, Pascal was the next dominant programming language designed for novices learning to program. Yet, when Brown studied error messages in Pascal, the messages were found to be inadequate [35, 36]. In fact, Chamillard & Hobart listed their concerns over syntax errors as a component of their motivation for their transition from Pascal to Ada97 in teaching programming at the US Air Force Academy [39]. du Boulay also cited Pascal's lack of meaningful error reporting as a drawback for using it as a teaching language [57]. In one of the early influential works on "enhancing" error messages (discussed in detail in Section 7) Schorsch introduced CAP (Code Analyzer for Pascal) [182]. CAP provided automatic, user-friendly feedback on syntax, logic and style errors in Pascal programs, and also provided enhanced error messages as the stock Pascal messages were often deemed insufficient.

In the intervening decades educators have continued to struggle with the choice of teaching novices in a language designed for beginners, or teaching an industry-strength language designed for professionals [22, 191]. Additional new languages such as Smalltalk

and Haskell have been developed with a focus on teaching, while other teaching languages have been derived from industry-strength languages from Assembly and C to Lisp and Scala. Over this time, improvements have also been made in both integrated development environments (IDEs) and in the development of completely new languages for novices.

COBOL is an early example of a language that was not designed for education, but COBOL statements have a prose-like syntax in order to be somewhat self-documenting. Litecky & Davis investigated error messages in the COBOL language, determining that their feedback was not optimal for programmers, and particularly challenging for novice programmers [120]. They found, like subsequent studies [14, 18, 164], that the distribution of error types encountered by students was skewed and therefore proposed that compiler writers should focus their optimisation efforts on the most common error types. The title of Kummerfield & Kay's paper investigating error messages in C, "The Neglected Battlefields of Syntax Errors", gave insight into the growing importance and concern over the area [110] as did Denny, Luxton-Reilly & Tempero's "All Syntax Errors are Not Equal" [51]. Though not designed for teaching, C++ became a dominant teaching language and Bergin, Agarwal, & Agarwal pointed out numerous issues with the C++ language in its use as a teaching language, many of them to do with the complexity of the compiler error messages [25].

Error messages present an interesting and somewhat unusual pedagogical situation. They have qualities that would seem to be positive for learners, as the feedback supplied by the machine is relatively immediate, consistent, detailed, and informative [175]. However messages can be of questionable value to students when they are phrased using highly technical jargon which may be of relevance to expert users, but can be confusing to novices [153, 171] Today, many IDEs offer features that (may) facilitate learning regardless of their initial purpose, such as providing visual clues such as highlighting and/or underlining to identify certain syntax errors prior to compilation. The "cosmetic" [56] and more functional [101] presentation aspects of programming error messages are the subject of current research but is an under-explored area.

When the Blue and BlueJ IDEs were developed specifically for learners, three main principles were at the core of the development: visualisation, interaction, and simplicity [108]. Block-based languages make certain structural errors in programs simply impossible due to the language specification or environment itself while other graphical languages such as LabVIEW make certain errors such as poorly constructed conditional tests impossible. Today, some IDEs offer annotated hints or even corrections generated by either an algorithm and/or by crowd-sourcing (see Section 7), but many IDEs still do not offer such feedback.

The level of exactness demanded by a compiler is arguably unique to the world of computing education. Gries illustrated this succinctly as early as 1974, [77, p83] saying, "Programming requires exactness and precision unknown in many other fields. Even in a mathematics paper, syntax errors and many logical errors can be understood as such and mentally corrected by the reader. But a program must be exact in every detail". This requirement for exact precision has been shown to be a common source of frustration for novices [176, p157]. In 1998, Lewis & Mulley wrote, "It is our belief that production compilers do not always address the needs of the student engaged in either learning a language or learning to program" [117]. Jadud completed a detailed study of student mistakes in Java and how students worked with the compiler to solve them. In doing so, he developed a metric called the error quotient to determine how well or poorly a student fared with syntax errors while learning to program [98], and this metric has been further studied and leveraged by others [16, 20, 92]. Other metrics based on programming errors (often utilising messages) followed such as Watwin [206], NPSM [38], and RED [16]. Linkages have also been identified between error messages and performance in programming [196].

The monitoring of novice programmer behaviour and mistakes while programming has a long history. Dy & Rodrigo analysed compilation logs to determine the errors most frequently encountered by novice programmers, and then modified their system to deliver what they believed to be more informative error messages [59]. Ahmadzadeh *et al.* studied novice error frequencies and debugging behaviours [2], Ettles *et al.* [66] & Rigby *et al.* [173] both classified various kinds of logic errors made by novice programmers with the goal of informing teaching practice, and research by McCall suggested that error messages have an imperfect mapping to student misconceptions [138]. Jackson *et al.* identified the most frequent errors among their novice programming students and confirmed a discrepancy between faculty identified errors and those errors novice programmers were encountering [96]. Such empirical research is necessary, as educators often have inaccurate intuitions regarding the kinds of errors that students are likely to make when learning to program. For example, Brown & Altadmri compared a study of more than 900,000 student users with the results of a survey of educators to investigate the understanding of which mistakes students make most frequently while learning to program in Java. They found that educators' estimates neither agreed with one another nor with the student-generated data [31].

While some authors had hypothesised that novice programmers do not read error messages, Barik *et al.* studied this directly [12]. They used eye-tracking software with undergraduate and graduate students as they attempted to resolve common errors in a Java code base, finding not only that participants do read error messages, but they found the difficulty of reading these messages to be comparable to the difficulty of reading source code, and impacted performance. They summarise with "The results of our study offer empirical justification for the need to improve compiler error messages for developers" [12, p575].

Educational programming systems have been increasing in number in more recent years. In 2015, Kölling wrote, "More systems of this kind have been published in the last few years than ever before, and interest in this area is growing" [108, p5]. Kölling attributes this to the rise of teaching programming in ever-younger age groups. In discussing the motivation for the development of Blue and BlueJ, Kölling says, "As Pascal before us, we had goals motivated by pedagogy: we wanted a clear and consistent representation of programming concepts, clear and readable syntax, good error messages, little redundancy, a small language core, and good support for program structure" [108, p11].

Measuring and improving the effectiveness of error messages for novices is of great interest to educators and has been the focus of much prior work. The SIGCSE 2011 best paper award went to

Marceau, Fisler, & Krishnamurthi for their paper entitled "Measuring the Effectiveness of Error Messages Designed for Novice Programmers" [132]. They defined a rubric that researchers can use to measure whether student code edits in response to displayed error messages reflect understanding of those messages. Applying the rubric to data from one course revealed that for many types of errors students responded poorly to the shown message. In 2014 Denny *et al.* found that enhancing programming error messages may not be effective [50] but a year later Becker found evidence that there can be positive effects [14]. Much work has taken place since then with no clear consensus. In Section 7 we comprehensively review work in this area, both historic and current.

In their important work, "Unlocking the Clubhouse", Margolis & Fisher found that more women than men transferred out of undergraduate computer science degrees before the third year [70]. Although student statements suggested this attrition was due to a loss of interest in the subject, the authors observed a more complex process stemming from a drop in confidence, stating "women and other students who do not fit the prevailing norm are disproportionately affected by problems like poor teaching, hostile peers, or unapproachable faculty" [70, p140]. Given this, it might seem plausible that interaction with unfriendly tools, which issue cryptic and unhelpful messages, might contribute to issues around student confidence.

Nonetheless, very little seems to have been published that more directly examines the particular impact of error messages on women and other underrepresented groups. Denny *et al.* found no gendered differences when they investigated level of engagement of students in their use of a tool that provides support for testing as well as drill and practice [52]. A multi-institutional, multinational 2016 study by Bouvier *et al.* found no significant difference in performance on different novice tasks – including between genders – on the number of compiling and non-compiling submissions [28]. Further exploration of the impact of programming error messages on student confidence, broken down by self-identities, could prove to be an important area for future work.

The issues novices face when dealing with compiler error messages was articulated well in 2004 when Ko [104, p206] wrote:

> Invisible rules are difficult to show. To overcome coordination barriers, learners must know a programming system's invisible rules. Today's systems lack explicit support for revealing such rules, merely implying them in error messages. Textual programming interfaces are limited. To avoid use barriers, the feedback and interactive constraints of every programming interface must be carefully designed to match its semantics. The textual, syntactic representations of today's systems make this goal difficult to achieve.

Unfortunately, the situation has not sufficiently changed to serve as a remedy in the intervening years.

In concluding this section we feel it appropriate to remind the reader that programming error messages have real effects on real people. The authors have seen anecdotal evidence at their home institutions, as well as documented evidence in the literature [129], that programming error messages are a contributing factor to real students leaving computing majors. It is also plausible that messages which are difficult to interpret for fluent English speakers may present even greater barriers for those who are non-fluent. Indeed, recent research that mined the BlueJ / Blackbox database revealed significant, albeit small, differences in error distributions between native language groups [170].

6 TECHNICAL

In order to generate effective error messages, a compiler must do two broad things:

(1) Detect that an error has occurred, and;
(2) Gather data about the state of compilation in order to craft an appropriate error message.

In some cases, *e.g.*, a type mismatch such as adding an integer to a list in Python as in: `1 + [1, 2, 3]`, this is a simple process where detecting the error is straightforward (the addition is erroneous) and the data necessary to generate a helpful message is readily available (`1` is an integer and `[1, 2, 3]` is a list). However, in other cases detecting an erroneous condition is difficult. It is true however that in some cases detecting an erroneous condition is easy, but obtaining the data to generate an effective message is difficult.

In analysing the literature, we have identified several general problems that impede the generation of effective error messages:

(1) *The completeness problem*: detecting all erroneous program behaviour in a (sufficiently powerful) programming language is undecidable.
(2) *The locality problem*: errors are frequently detected far away from their generation sites.
(3) *The mapping problem*: errors are detected in representations that do not cleanly map back to the original code.
(4) *The engineering problem*: rich error handling leads to less maintainable and less performant code.
(5) *The liveness problem*: in certain situations, compilation tools are given partially-completed programs as input which they are not traditionally designed to handle.

To understand these technical challenges, we first describe these problems in detail and then we briefly explore how researchers and compiler implementers have addressed these challenges.

6.1 Challenges

6.1.1 Completeness of Analyses. Generating effective diagnostics poses significant technical challenges. Rice's theorem states that reasoning about any nontrivial property of a program's behaviour is undecidable [172], so compilers tend to focus on restricted classes of errors such as improper syntax and violation of typing rules. In general, a program analysis may have one of the following properties but not both:

- The analysis may be *sound*, meaning that if the analysis; or claims a program is free of errors, then it really is.
- The analysis may be *complete*, meaning that if the analysis claims a program is erroneous, then it really is.

Type systems in a programming language demonstrate the trade-off between soundness and completeness. The primary goal of a type system is to detect potential programming errors before running the code itself. This can be done by analysing the code at compile-time (*static type checking*), and if the compiler determines

the program is free of type errors, then it will not encounter one when run; thus, the analysis is sound. On the other hand, the type of a value may be checked at runtime prior to performing an operation on it that requires the value to be of a specific type (*dynamic type checking*). This will only detect errors that occur in a specific program execution, so an error is only reported if it is actually present; thus, the analysis is complete. Static type checking reports errors earlier, at the cost of possibly disallowing meaningful code, while dynamic type checking detects only those errors that occur when the program is executed, missing errors that may be exposed by different inputs or execution paths. In practice, many languages employ a combination of static and dynamic type checking.

More complex static type systems enable more properties to be checked at compile time. However, they impose additional burdens on programmers over simpler type systems. To reduce the programming effort, languages may support *type inference*, where types are inferred automatically by the compiler based on how a particular variable or object is used. Unfortunately, this can result in confusing error messages when the inference fails, such as in the following OCaml code [116, p3]:

```
let map2 f aList bList =
    List.map (fun (a, b) -> f a b)
             (List.combine aList bList)

let lst = map2 (fun (x, y) -> x + y) [1;2;3] [4;5;6]

let ans = List.filter (fun x -> x==0) lst
```

This produces the following error in ocamlc 4.08:

```
File "main.ml", line 5, characters 30-35:
5 | let lst = map2 (fun (x, y) -> x + y) [1;2;3] [4;5;6]
                                  ^^^^^
Error: This expression has type int but an expression
was expected of type
         'a -> 'b
```

The problem is that the `map2` function requires as its first argument a function that takes two arguments in curried form, but it is provided one that takes two arguments as a pair. The correct call is as follows:

```
let lst = map2 (fun x y -> x + y) [1;2;3] [4;5;6]
```

Not only is the error incorrectly localised, it also does not provide sufficient guidance for fixing the error to the programmer. Both problems are artefacts of the algorithm used for type inference.

In addition to errors that strictly violate the type system, some compilers also generate warnings about common programming errors, such as failing to initialise a variable or performing an implicit type conversion in a context that is likely to be erroneous. Prior work has also explored more advanced heuristics for detecting likely errors [87, 203], at the cost of generating more false positives.

6.1.2 Error Localisation. The localisation of errors is a particularly difficult problem. Syntax errors such as missing semicolons or curly braces may be detected at a later point than the source of the error, and error-recovery algorithms in parsers can further confuse users by producing messages referring to spurious errors.

The following is a Java program that exemplifies the difficulty with localising syntax errors:

```
class Main {
  public static void main(String[] args) {
    if (args.length > 0) {
      int sum = 0;
      for (int i = 0; i < args.length; i++) {
        sum += Integer.parseInt(args[i]);
      System.out.println("Sum: " + sum);
    }
    System.out.println("done");
  }
}
```

The OpenJDK 11 compiler reports the following error:

```
Main.java:11: error: reached end of file while
 parsing
}
 ^
1 error
```

In addition to the error message being particularly uninformative, the reported line number is the last line of the file, while the source of the error is likely on a previous line. In fact, the program can be corrected by inserting a closing brace before line 7, after line 7, or at line 11, each of which results in different program behaviour at runtime. The compiler does not know the programmer's intent, so it only identifies an error when it reaches the end of the file without encountering the expected closing brace.

Languages with sophisticated type systems also pose a problem for localisation; type-inference rules can be complicated and non-local, often resulting in inscrutable error messages that are far from the source of the error, as in the OCaml example above. C++ template instantiation is another canonical example, where misuse of library templates (*e.g.*, attempting to insert a vector into an output stream) produces errors in library code that the user did not write.

As an example, the following is an erroneous C++ program:

```
#include <algorithm>
#include <list>

using namespace std;

int main() {
  list<int> items = { 3, 1, 4 };
  sort(items.begin(), items.end());
}
```

The code populates a list of integers and attempts to sort it by invoking the library function template `sort` on the begin and end iterators of the list. However, the code does not compile, and GCC 7.3 produces the error message in Figure 3. The compiler reports an obscure error within library code, and it proceeds to report every overload of `operator-` that it found and why each one does not work. This is because the call to `sort` causes the function template to be instantiated with list iterators (`std::_List_iterator<int>` in the error message), and the generated code is then checked for errors. The code for `sort` applies the subtraction operator to the given iterators. However, list iterators do not support subtraction, so the compiler reports that it could not find an overload of the subtraction operator that works on list iterators. The true source of the error is that the programmer violated the requirements on

```
In file included from /usr/include/c++/7/algorithm:62:0,
                 from list.cpp:1:
/usr/include/c++/7/bits/stl_algo.h: In instantiation of 'void std::__sort(_RandomAccessIterator,
 _RandomAccessIterator, _Compare) [with _RandomAccessIterator = std::_List_iterator<int>;
 _Compare = __gnu_cxx::__ops::_Iter_less_iter]':
/usr/include/c++/7/bits/stl_algo.h:4836:18:   required from 'void std::sort(_RAIter, _RAIter) [with
 _RAIter = std::_List_iterator<int>]'
list.cpp:8:34:   required from here
/usr/include/c++/7/bits/stl_algo.h:1969:22: error: no match for 'operator-' (operand types are
 'std::_List_iterator<int>' and 'std::_List_iterator<int>')
     std::__lg(__last - __first) * 2,
               ~~~~~~~^~~~~~~~~
In file included from /usr/include/c++/7/bits/stl_algobase.h:67:0,
                 from /usr/include/c++/7/algorithm:61,
                 from list.cpp:1:
```

Figure 3: Error message from GCC 7.3. The generating source is in Subsection 6.1.2.

`sort`, but the compiler does not know those requirements and can only reason about the resulting instantiated code.

6.1.3 The Mapping Problem. Code transformations also complicate error messages; errors may be detected in the transformed code, with messages that refer to that code rather than the original, pre-transformed code. Code transformation is supported at the program level by languages with macros, such as Scheme, Haskell, and C/C++. With *embedded domain-specific languages (EDSLs)*, programmers write code in a host language using the abstractions provided by the EDSL library, and the code passes through the host compiler before being executed or further transformed by the EDSL. Mapping back to the original code is therefore a significant challenge when reporting errors [55, 88, 152].

The following example demonstrates the mapping problem in R5RS Scheme:

```
1 (do ((vec (make-vector 5))
2       (0 (+ i 1)))
3      ((= i 5) vec)
4   (vector-set! vec i i))
```

The program has a typographical error on line 2, missing the identifier `i` before the initial value of 0. The `plt-r5rs` interpreter reports the following error:

```
do.scm:2:6: let: bad syntax (not an identifier)
  at: 0
  in: (let doloop ((vec (make-vector 5)) (0 (+ i 1)))
   (if (= i 5) (begin (void) vec)
   (begin (vector-set! vec i i) (doloop vec 0))))
  location...:
   do.scm:2:6
```

The canonical definition of a **`do`** loop in the R5RS specification is a macro that transforms it into a **`let`** expression. The `plt-r5rs` interpreter applies this transformation, resulting in an erroneous **`let`**, and the error message refers to the transformed code rather than the original program source code.

While the example above can be addressed by incorporating direct support for the **`do`** syntax in the Scheme interpreter, user-defined macros pose the same mapping problem, and the interpreter does not have *a priori* knowledge of such macros.

6.1.4 Engineering Challenges. Improving error messages takes significant engineering effort, and compiler developers generally prioritise new features over improvements to error handling [202]. Incorporating better diagnostics can also lead to compiler code that is less performant and maintainable; for instance, Lewis & Mulley observed a 30% reduction in performance even after significant optimisation in their Modula-2 compiler [117]. For syntax errors, taking advantage of parser generators that produce higher-quality error messages [40, 156] requires a language's grammar to be reformulated in the restricted forms required by those generators.

At a higher level, compiler developers are very well-versed in both the specification of the language as well as the internals of the compiler. They therefore often design error messages in these terms [202]. Additionally, compilers in general reason about programs in a different manner than most programmers [9]; the compiler's primary task is to produce object or byte code, and it often reports errors only as a byproduct of this process [57]. Finally, compiler-development teams do not often (enough) include experts in human computer interaction [202].

In Section 7 we discuss improving programming error messages *after* they are generated by the compiler, which is a very different endeavour to designing better messages from first principles. The latter can take advantage of the information available during compilation, while the former works with just the input and output of the compiler.

6.1.5 Live Compilation. Classically, the compiler is fed a complete program as input and compilation ends immediately when an error is detected. We traditionally call such scenarios *batch compilation*. However, with the advent of integrated development environments (IDEs), it is commonplace that *partial* programs are given to the analysis tool instead. And in contrast with the classical scenario, the programmer desires that the tool processes as much of the available program as possible even though an error might be found early

during checking. For example, a tool that provides auto-completion will need to type check a program that is still being edited to provide its information. In contrast with batch compilation, we call such scenarios *live compilation* to emphasise that the code is still being edited during analysis.

Live compilation introduces an additional pair of problems on top of those that we have already raised that can prevent the generation of effective programming error messages. One of these concerns is coping with *partial* programs. Traditional language analysis algorithms, *e.g.*, parsing and type checking, assume that the program is complete. However, for a partial program, parsing will fail to produce a complete AST so that type checking never happens.

This leads to undesired behaviour in many common scenarios. For example, consider the following partial Java program:

```
public class Program {
  public static void main(String[] args) {
    String input = args;    // Type mismatch here
    for (char c : input) {
      // Editing cursor here
  }
}
```

Here the partial program is grammatically correct save for the missing contents of the for-loop. There is a clear type error here—`args` has type `String[]` whereas `input` has declared type `String`—but a traditionally-built compiler will never type check the program to discover this fact. Instead, the compiler will report an unhelpful parse error regarding the missing curly brace for the for-loop.

The other concern is *incremental processing*. In batch compilation scenarios, invocations of the compiler are independent and share little-to-no information. However, in live editing scenarios, the programmer adds onto existing code that has already been processed in a structured manner. For small programs, it is fine to analyse the code from scratch. However, for larger projects, re-analysis quickly becomes an infeasible proposition.

6.2 Current Research

The challenges described in Section 6.1 cut across the entirety of the compilation pipeline. However, the work addressing these issues is often contained within individual parts of the compilation pipeline. Here we sample this work to demonstrate how these challenges are being addressed in the literature. We restrict ourselves in this section to approaches that address these challenges directly, focusing primarily on techniques within compilers, interpreters, and analysis tools; Section 7 will discuss in detail work that enhances programming error messages external to these program analysis tools.

6.2.1 Parsing. Work on improving error messages in the parsing phase reaches back over five decades. For example, Gries described a parser that incorporated common errors and recovery actions into the parsing table [76]. Commonly used parser generators support error productions to enable recovery from parsing errors, avoiding spurious errors after an initial one [115, 210]. Many other schemes have been implemented for error recovery, with the common goal of maximising the set of true errors detected while minimising false positives [40, 57, 75, 100]. More recent work has focused on improving the quality of error messages rather than the quantity [89, 112], and on automatically correcting syntax errors [60, 100, 205].

Production compilers usually attempt to maximise the number of errors found. However, this is not necessarily ideal for novice programmers, who are often better served by focusing on just the first error [21, 57]. Compilers such as GCC and Clang include options to configure the limit on the number of errors reported.

6.2.2 Type Inference. Several techniques have been used to improve the quality of error messages in the presence of type inference. A common approach is to improve type-inference systems to better localise type errors, such as by modifying algorithms to avoid left-to-right bias, making use of a global constraint graph, or otherwise increasing the amount of information tracked by the inference engine [63, 79, 86, 87, 114, 137]. There has also been recent work to develop more sophisticated constraint systems to perform inference in complicated type systems, identifying the actual program locations that cause a type error [168, 169, 217]. These algorithmic improvements lead to better localisation as well as higher-quality error messages. Type checkers may also employ heuristics to identify the true source of an error, as well as to suggest possible fixes [62, 79, 87]. Other systems allow a library to specify custom type checking for code that uses the library [86].

6.2.3 Metaprogramming. Recent versions of C++ have taken steps toward addressing the locality and mapping problems in the context of metaprogramming. C++11 introduced static assertions and type traits, allowing library writers to customise error messages if a type requirement is not met [94]. C++20 will contain concepts and requires clauses, incorporating better constraint checking into the template instantiation system [95]. As an example, adding a requirement to the standard `sort` function template that the arguments must be random-access iterators should enable the compiler to produce a better error message than that in Figure 3, *e.g.*, by reporting that a list iterator is not a random-access iterator.

EDSL frameworks often provide mechanisms for detecting errors pre-transformation, avoiding the mapping problem and providing better error localisation. The Proto framework for defining expression templates in C++ provides a meta-function for matching an EDSL expression against a grammar, producing a readable error message in the case of a mismatch [152]. Another approach is to modify the type checker of the host language to apply specialised type rules to code written in the EDSL [184]. Finally, EDSLs themselves can be architected to perform error detection directly [55].

6.2.4 Runtime Errors. As listed in Table 1 only 8 papers in our corpus were tagged as relating to "runtime errors". Only two of these are post 2010 and almost all are specific to a particular language or domain such as parallel computing [122, 123]. For an interesting historical account, the reader is guided to [204].

6.2.5 Cross-cutting Approaches. Some tools for improving error messages apply techniques that are agnostic to the specific compilation phase that produces an error. Often, these systems treat the compiler as a black box, working solely with the program source code, whether or not the compiler signals an error, and the error messages that the compiler reports. One such technique is to make incremental mutations to the program source code to identify what changes result in successful compilation, to improve localisation

of an error as well as to suggest possible fixes. This is similar to the concept of delta debugging [216], and it has been applied to improve the quality of type errors as well as to identify fixes for more general programming errors [29, 116, 159, 180]. Recent work has also applied machine-learning techniques toward localisation and suggested fixes, such as training language models on correct code [60], extracting features from erroneous code to train error models [213], and training neural networks on correct and erroneous variants of the same program [3, 78].

Much of the work on cross-cutting approaches has been done in the context of enhancing error messages through external tools, rather than directly within compilers or interpreters. Section 7 discusses these techniques in more detail.

7 ENHANCEMENT

Much work involving programming error messages in recent years (and the less active preceding decades) has been reactive. This is often due to researchers and practitioners witnessing the trouble that many students have with these messages, possibly having had a similar experience themselves, and seeking to alleviate these problems by making standard messages better by 'enhancing' them. Providing more detailed error messages to students has been identified as a strategy to combating the difficulties faced when learning programming [165]. However, this does not imply that longer, more verbose messages are better, and with more information comes the risk of misinterpretation on the student's part and genuine error on that of the system – for instance when the system suggests fixes that are not guaranteed (which is normally the case). This is a deceptively complex area to conduct effective research in, largely because there are three components: the user, the system, and the message itself. More specific difficulties come about because such research:

- can span almost all programming languages;
- can span almost all programming environments;
- is closely intertwined with other linguistic and environmental features (e.g. some languages may have more usable messages than others; some environments may present these messages via differing mechanisms with differing effects)
- spans several computing domains including technical, pedagogical and human computer interaction; and
- must take into account fundamental domains such as: human psychology (message 'tone' and presentation, etc.), education (interpretation of feedback) and natural language (programming error messages are by definition a mix of natural and programming language, and their readability – from a fundamental perspective – is important).

7.1 What is Message Enhancement?

What exactly constitutes enhancement of a programming error message, and how to properly refer to it, is complex. In Sections 2.2 and 3.3 we laid out our definition of "message" for the purposes of this study as text-based messages emitted by a compiler or interpreter (often passed on by an environment/IDE/editor) in response to a programmer-committed error. "Error" itself is defined in Section 3.2.

As such we consider enhancement to be altering the text of a message for the purpose of improving the usefulness of the message in terms of aiding a human in fixing the error that generated it. This excludes underlining code, icons alerting the user to an error or message, etc. Even with a relatively tight definition however, there are grey areas. For instance BlueJ (up to and including version 3) only presents one message at a time (textually), even if multiple messages were emitted by the compiler. Whether or not this is considered enhancement is debatable. Although not necessarily altering a given single message, this mechanism does alter the text of a number of messages (by excluding the text of all but the first). Other environments (including BlueJ 4) sometimes concatenate messages (but otherwise do not alter them) for instance, so they fit neatly in a pop-up window.

Although "enhancement" is a commonly used term, particularly in recent years, and especially when dealing with the *text* of programming messages [18, 50, 158, 163], "enhancement" and variants such as "enhance" are not universally used. For instance, "improved" has been used by a small number of recent papers [106]. Other terms include "supplemental" [41, 200]. The programming languages community also uses other terminology: LLVM[7] calls them "expressive diagnostics", while Rust[8] has "explanatory error messages" and "extended error messages"[9].

For the purposes of our work, and consistency with prior work, enhancement refers to the action of a tool, to improve the presentation of errors as presented by another tool such as an IDE or a compiler. For instance, Becker designed and utilised a Java editor that enhanced the text of standard javac error messages [15] and Karkare (with Umair *et al.*) have worked with C [4]. Similarly, Automated Assessment Tools (AATs) have been designed that enhance messages. For instance, Denny *et al.* with Java [50], while Prather *et al.* [163] and Pettit *et al.* [158] have worked with AATs that utilise C++.

Many modern tools and environments do more than just 'relay' message text from the compiler to the user. Regardless of whether the text is enhanced, there are many other ways of presenting messages that could be considered to be enhancement. Examples include pop-ups with message text as in BlueJ 4, icons such as Eclipse's 'lightbulbs', that provide further "supplemental" information, and even suggestions on how to resolve the issue causing the message, although in some platforms these can be misleading [15]. For the purposes of this paper we limit the definition of enhancement to the modification of the text emitted by the compiler. This is for several reasons, but two primary ones:

(1) that is where the bulk of the work, both recent and historical, has been focused ; and
(2) going beyond text modification opens up an almost limitless array of features and actions that could possibly be considered enhancement. As discussed above, underlining, highlighting, pop-ups and other mechanisms are becoming quite common. Interestingly, such features are rarely studied but there is some interest in this [101].

[7] http://clang.llvm.org/diagnostics.html
[8] https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html
[9] Much of this discussion on terminology was informed through personal correspondence with the first author of [10]

It is important to note that one related area that may be confused with error message enhancement is finding/correcting/studying (often syntax) errors (not programming error *messages*) in code. Some researchers believe this to be a more fruitful avenue. For instance Kölling & McCall just this year stated: "Various past studies have analysed the frequency of compiler diagnostic messages. This information, however, does not have a direct correlation to the types of errors students make, due to the inaccuracy and imprecision of diagnostic messages." [140, p38:1]. Yet, as we have seen in this report, much work has focused on messages. In fact, "improve automatically generated diagnostic messages" is listed as future work by [140, p.38:22].

Blurring lines even further, some tools that find errors in code provide feedback in the form of messages (for instance, see [26]). However, such messages *originate* in a tool/analyser external to the compiler/interpreter. As these messages do not originate in the compiler/interpreter, and are not therefore modified versions of the messages produced by them, we do not consider them to be enhanced programming error messages.

Regardless, going 'beyond text' is beyond the scope of this paper which is aimed at providing an overview of the work that has been done (which focuses on message text), and to provide a set of guidelines for message design where again, most prior work refers to text-based messages. In Section 7.4 we briefly discuss possible short-term future trajectories before touching on wider (beyond enhancement or beyond text) directions in Section 7.5.

7.2 A Brief History of Message Enhancement

7.2.1 Pre-2000: Sparse Activity. Although work on error message enhancement goes back over half a century – the earliest paper we found on the topic was from 1965 [177] – out of the 107 papers we tagged as *enhancement*, only 10 were pre-2000. Another 31 were from 2000-2009, and the remaining 66 were from 2010-present, with 41 from 2015-present. Figure 4 shows this growth on a decade-by-decade basis.

The earliest paper on enhancement we found (and also the oldest in our combined corpus) dating from 1965, was on PUFFT (Purdue University Fast FORTRAN Translator) a translator for FORTRAN

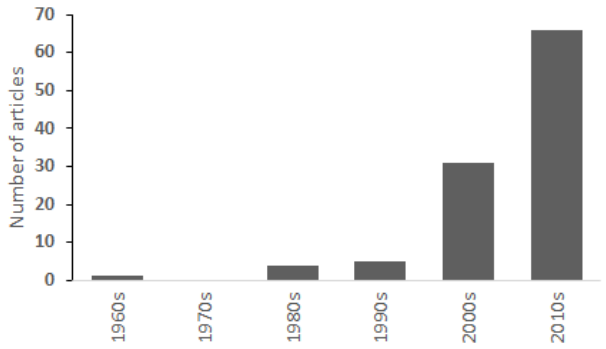

Figure 4: Number of articles per decade tagged as "Enhancement", appearing in both our original corpus and found through quasi-systematic search.

IV which included a "rather elaborate diagnostic message writing routine" [177, p661]. DITRAN (*DI*agnostic FOR*TRAN*), an implementation of FORTRAN with rather extensive error handling capabilities [146] soon followed. Interestingly, like PUFFT, DITRAN was not a tool that performed enhancement as became prevalent in the time between then and now ("intercepting" programming error messages and enhancing them after they had been generated) but an implementation of FORTRAN. It was therefore more similar to very recent efforts in designing new languages that have error messages in mind from the design phase, *ab initio*, despite the fact that DITRAN was reactive – in as much as it was trying to improve upon FORTRAN – it was not intended to be an entirely new language. (This is yet another grey area of programming error message enhancement.) Nonetheless, DITRAN had a repertoire of 300 messages. As discussed in Section 5, FORTRAN was not designed for teaching, but was used for teaching, especially early in terms of computing education. In fact, PUFFT and DITRAN were created with students in mind, as were WATFOR (*WAT*erloo *FOR*TRAN) and WATFIV (WATFOR's successor) which also included an impressive number of custom error messages [43]. A full listing of WATFOR and WATFIV messages are presented in [43].

In 1987 Kantorowitz & Laor developed a 'syntax error handling system' for Pascal that was "particularly good at avoiding misleading messages" [100, p632] in student code. In 1995 Schorsch introduced CAP (Code Analyzer for Pascal) [182], the first enhancement effort for a teaching language that we encountered. CAP provided automatic, user-friendly feedback on syntax, logic and style errors in Pascal programs, and also provided enhanced error messages as the stock Pascal messages were often deemed insufficient.

Despite a small number of studies, those that do exist pre-2000 are predominantly small-scale and often the evidence they present is anecdotal in nature. Nonetheless many are fairly monumental achievements in terms of the work required – such as implementing FORTRAN interpreters largely or entirely from scratch.

7.2.2 2000-2009: Still Largely Anecdotal. Fuelled by the internet and cheap storage, the ability to easily log large volumes of student data became possible. This led to increased attention on programming error messages in the first decade of this century. However heavy-weight efforts like those prior to 2000 did continue as well, although some technically do not meet our definition of enhancement. For instance in 2003, Hristova *et al.* developed Expresso, an 'error detection advisory tool'. Although Expresso was a preprocessor, it did have its own error messages that circumvented the compiler altogehter. Similar tools were developed around the same time, sometimes as part of larger systems. One such example is Dr. Scheme [69] (which became Dr. Racket [132]) and included language levels/subsets with customised error messages for each level/subset.

On the enhancement front, Flowers, Carver & Jackson developed Gauntlet in 2004, a system that explains the syntax errors students encounter while learning Java [72]. Thompson developed GILD, an IDE for beginner Java programmers, which provided 51 "supplemental" error messages [200]. Coull presented a support tool called SNOOPIE (Supporting Novices in an Object Oriented

Programming Integrated Environment). SNOOPIE, rather than replacing programming error messages geared for experts, presented supplementary messages more suitable for novices.

During this period a greater ease of access to data led to an increased rate of publication. However this also seems to have contributed to a large number of fairly trivial experiments presenting largely anecdotal evidence.

7.2.3 2010-2015: Increasing Empiricism, Conflicting Results. In 2011 Hartz developed CAT-SOOP, a tool which allows for automatic collection and assessment of various types of homework exercises in Python. The tool includes an error analyser that provides "simple explanations of common error messages in plain English" [84, p42]. The original error message generated by Python is still displayed, but is augmented by a simple explanation of what the error message means, in the hope that students will begin to connect the simple explanation with the error message that the interpreter generates. This advice was put forward by Coull [41, 42] and later followed by Becker *et al.* [15, 18] who developed a Java editor called Decaf that intercepted and re-worded 30 of the most frequent Java error messages. Empirical evidence from a control/treatment study involving hundreds of students and over 50,000 programming error messages showed that Decaf reduced student error frequency, and reduced indications of struggling students. A year prior to this, Denny *et al.* provided enhanced error messages in the Automatic Assessment Tool, CodeWrite [52], and found that enhancing compiler error messages was ineffectual [50]. However a closer analysis found that directly comparing the Becker *et al.* and Denny *et al.* studies yielded inconclusive outcomes, the most conclusive being that comparing (even very similar) studies in this arena is quite challenging.[10].

7.3 Current Results in Enhancement

The Decaf editor was used in several more studies, most showing statistically significant but not very overwhelming support for enhancing compiler error messages [15, 18–20]. These studies spanned a few different contexts including semester-long 'free-range' programming in addition to a controlled programming quiz [19]. Simultaneously a few other studies developed along similar lines, but often as automatic assessment tools as opposed to editors. Pettit *et al.* tackled recent conflicting evidence head-on and came up none-the-better with "Do Enhanced Compiler Error Messages Help Students? Results Inconclusive" [158]. That study utilised Athene, an automated C++ assessment tool. On another hand, Harker used Decaf independently of Becker and had similar results, partially replicating them [80]. It should be noted that replication work in computing education is not performed as much as it should be [1, 49, 155, 219] and the area of error message research is no exception. Prather *et al.* used the same software as Pettit *et al.* [158] but a very different think-aloud protocol and found results similar to Becker in that there seems to be weak positive effects associated with enhancing compiler error messages. In 2017 and 2019 Kohn, investigating enhanced Python programming error messages, found mixed results as well [105, 106].

It is likely that much of the recent (post-2000) interest in programming error message enhancement can be attributed to continually improved technical capabilities in logging student data, particularly compiler activity and source code (not just programming error messages) on ever-growing scales [190]. This, coupled with a lack of tangible results in the area and increased rigour in computing education research, culminated in an explosion of studies (keeping in mind the context), many presenting empirical evidence on the effectiveness of error message enhancement. For instance, the Blackbox project has been logging compiler activity of Java programmers using the BlueJ pedagogical editor [109] and as of 2018 has logged over 306 million compilation events including all error and warning messages [32]. As of March 2018, 19 primary Blackbox studies had been published [32]. Five of these relied solely on Blackbox error messages with two looking at the content of the messages.

Additionally, and quite recently, language and compiler designers have shown an interest in providing better error messages from first principles with languages such as Elm[11] and the Clang project[12] putting dedicated effort into the effectiveness of diagnostic messages. Additionally, Quorum is in the beginning phases of doing the same[13], and the most recent version of GCC (version 8)[14] has had some error messages improved, for instance to provide more accurate error location. GCC 8 also provides some 'fix-it hints' so that IDEs can offer to automate the fix. It is possible that some of this is a reaction to the growing base of work on 'fixing' compiler error messages through enhancement, a growing focus on human computer interaction and user experience issues, natural progression, or a combination of these. Regardless, these efforts are certainly viewed as a positive step in terms of language and compiler design.

7.4 Into the Future

It is unclear what direction programming error message enhancement will take. There are however extremely promising trajectories developing in related areas that might end up rendering enhancement unnecessary which is surely a better solution, should it be achievable. After all, enhancement is a post hoc fix by definition. Surely there must be a better way – "It is unreasonable to think that enhancing compiler error messages will completely alleviate the problems students have with them" [14, p81].

In 2012 Watson, Li & Godwin introduced BlueFix, an online tool integrated into BlueJ that uses crowd-sourcing to assist programming students with error diagnosis and repair, with greater effect than standard programming error messages [207]. Working along similar lines but several years later, Thiselton & Treude have proposed Pycee, a plugin integrated with the Sublime Text IDE that provides enhanced programming error messages for Python programmers [199]. The enhanced messages are constructed by automatically querying Stack Overflow. Also utilising Stack Overflow, Wong *et al.* [211] explored a methodology to automatically extract a corpus of syntax errors and their fixes, data that could be used to develop better messages.

In 2019, Ahmed *et al.*, building on prior work [3] published a paper on arXiv detailing a system called TEGCER that also provides automated feedback in lieu of standard programming error

[10] https://cszero.wordpress.com/2016/11/18/you-are-what-you-measure-enhancing-compiler-error-messages-effectively/

[11] https://elm-lang.org/blog/compiler-errors-for-humans

[12] https://clang.llvm.org/diagnostics.html

[13] https://quorumlanguage.com/evidence.html

[14] https://developers.redhat.com/blog/2018/03/15/gcc-8-usability-improvements/

messages. The system uses supervised machine learning models trained on more than 15,000 error-repair code samples and has a stated accuracy of 97.7% across 212 error categories [4]. Both of these approaches are claimed to outperform DeepFix [78]. Another avenue showing promise are feedback hints and alternative means of helping students correct errors. For instance, the system presented by Marwan *et al.* provides "expert-authored help messages, often in response to specific errors in student code" [134, p521]. These works and others may hold promise to possibly augment, or perhaps eventually eliminate the need for programming error message enhancement.

7.5 Beyond Enhancement

An often overlooked fact is that there are situations and environments where (at least syntax) error messages need not occur at all. In 1983 Brown noted "…some programming environments remove the need for *certain* error messages altogether" [36, pp248-249] (emphasis ours), citing the Cornell Program Synthesizer [197] and the Interlisp system [198]. Interlisp was an environment based on Lisp and "for experts" [198, p26] while the Cornell Program Synthesizer was a syntax-directed environment for students.

However, environments that are popular today for children successfully eliminate the need for (text-based) programming error messages. For instance most block-based environments such as Scratch avoid text-based text messages altogether as explained by [130, p16:5-16:6]:

> When people play with LEGO® bricks, they do not encounter error messages. Parts stick together only in certain ways, and it is easier to get things right than wrong …similarly, Scratch has no error messages. Syntax errors are eliminated because, like LEGO®, blocks fit together only in ways that make sense. …Of course, eliminating error messages does not eliminate errors …[however] a program that runs, even if it is not correct, feels closer to working than a program that does not run (or compile) at all.

Although the jump from Scratch to the complexities of high-level industrial-strength languages such as Java might seem great, there is progress being made. For example, frame-based editing such as Stride [5] allows more complex programming with the elimination of at least some common syntax errors by design.

8 GUIDELINES

As discussed above, there is a long history of reporting on the difficulty of programming error messages. Running in tandem with that is a rich history of researchers suggesting design guidelines for improving those error messages (or writing better ones from scratch). Most of these suggestions, especially in the 1960s, 1970s, 1980s, and 1990s, were based on anecdotal evidence or expert opinion. Other guidelines, although based on thorough research, are not presented as such in the published literature[15], although they are sometimes leveraged in practice[16]. Modern software engineers have always had ideas about how to write better error messages and this trend continues up to the present [151]. However, these have slowly given way to a rise in guidelines that have been empirically validated. In the sections below, we have attempted to collect as many of these guidelines as possible and organise them into relevant groups, including anecdotal, but relying on empirical evidence. Throughout the decades of work on this topic, it appears that shifts in design mentality followed major shifts in industry tools (see Figure 5). Over the past 60 years, researchers have mostly suggested the same kinds of guidelines that language creators and maintainers should follow to make programming error messages more usable. This alone is interesting because, even though programming languages have drastically changed in that time period and countless new ones have appeared, it reveals that we mostly still have the same kinds of problems we had 60 years ago. However, particularly in the last decade as noted below, some newer languages and updates to old standard bearers alike have been starting to make efforts towards implementing some of these guidelines. Moreover, new research within the past year has brought about new directions for programming error message guidelines in areas such as cognition, rational argumentation, and timeliness of the presentation of error messages.

All of the papers in both our original corpus and literature search that were tagged as offering guidelines or pre-guidelines were analysed for the types of suggestions made. We found 10 general categories of guidelines that could be considered to be generalised guidelines. The papers were further classified as to the evidence presented for the guideline suggestions. Papers published prior to the year 2000 with an assumption of text based interfaces were classified as historical, and papers which attempted to provide experimental evidence to support their guidelines were classified as empirical. All other papers were classified as anecdotal. The intersection of guidelines espoused and evidence supported can be found in Table 2.

The majority of papers did not attempt to provide comprehensive guidelines, but rather focused on a particular set of suggestions. There were, however, four papers which attempted to provide overarching, concrete guidelines for error message creation: Horning (1976) [90], Schneiderman (1976) [187], Traver (2010) [202] and Barik (2018) [9]. We compare the generalised guidelines we extracted from our corpus to those proposed in these works. Out of 22 guidelines proposed by these four authors, our ten generalised guidelines encapsulate all but two as shown in Table 3.

In the remainder of this section, we analyse each of the ten guideline categories we found, reporting on their history, current state of development, and the level of empirical support for their adoption.

8.1 Increase Readability

Horning lists 'readability' as a criterion that good error messages should exhibit [90], but fails to provide any mechanism to assess readability. Authors have used a variety of terms to describe readability, such as saying that error messages must be 'comprehensible' [187], use 'plain language' [142] or be 'simplified', using 'familiar vocabulary' [132], and not be 'cryptic' [187] or use 'technical jargon' [142]. However, no authors have provided concrete advice as to how to achieve or assess readability in error messages.

[15] https://docs.racket-lang.org/htdp/#%28part._error-guidelines%29
[16] https://quorumlanguage.com/evidence.html

Table 2: Classification of papers by guidelines suggested, and evidence presented. The category "Historical" refers to papers published before the year 2000, "Empirical" to those providing experimental evidence, and "Anecdotal" to the remainder.

	Historical	Anecdotal	Empirical
Increase Readability	[37] [47] [64] [74] [93] [186] [187]	[71] [209] [215] [67] [72] [89] [90] [104] [124] [138] [142] [194] [201] [202]	[128] [145] [41] [106] [132] [133] [148] [150] [149] [154] [174] [214]
Reduce Cognitive Load	[37] [47] [146] [187] [208]	[67] [91] [100]	[128] [145] [9] [21] [106] [150] [149] [153] [163] [162] [207]
Provide Context	[36]	[123] [215] [42] [67] [69] [90] [91] [100] [104] [124] [201] [202] [212]	[11] [8] [9] [150] [153] [214]
Use a Positive Tone	[37] [47] [64] [93] [186] [187] [208]	[71] [209] [72] [89] [90] [91] [202]	[145] [148]
Show Examples	[93]		[128] [9] [110] [154] [159] [163] [199] [207]
Show Solutions or Hints	[64] [146]	[215] [42] [89] [100] [138] [142] [201] [202]	[128] [14] [83] [110] [132] [133] [148] [153] [159] [174] [178] [199] [207]
Allow Dynamic Interaction	[36] [37] [47]	[71] [42] [142] [194] [202]	[128] [9] [10] [159] [163] [199] [207]
Provide Scaffolding	[74] [208]	[42] [72] [201]	[11] [8] [10] [14] [132] [133] [163] [162] [207] [214]
Use logical argumentation			[9] [10]
Report errors at the right time		[69] [72]	[159] [178]

Readability metrics have been applied to source code, with little success [45].

There are multiple measures of readability for 'normal' prose, including the Fry Readability Graph (see Figure 6), Flesch formula, Dale-Chall formula, Farr-Jenkins-Paterson formula, Kincaid formula, Gunning Fog Index, and Linsear Write Index [118]. However, there are currently no papers that attempt to apply these measures to programming error messages.

There is clearly wide anecdotal (or, common-sense) agreement that readability is important, with approximately 50% of the guidelines papers discussing or mentioning readability in some way. While most of the papers don't feel the need to motivate readability as a guideline, Flowers *et al.* [72] mentions that readable messages are more memorable, and Traver [202] argues that writing more readable error messages promotes error correction by 'recognition rather than recall'.

Readability in various forms was included in 11 of the 25 papers with empirical evaluations of their guidelines, with several showing that systems which include more readable error messages had positive impacts on error rates and repeated errors [14], improved student satisfaction [214], and reduced frustration [174]. However, none of these systems assessed the effects of readability in isolation, nor did any paper formally define a rubric for evaluating readability.

8.2 Reduce Cognitive Load

Cognitive Load Theory (CLT) states that humans have a finite ability to efficiently process input based on working memory (intrinsic) and situational/contextual (extrinsic) factors [195]. Long before it was first proposed in the 1980s and then first discussed in the context of computing education in 2003 [185], researchers investigating programming error messages were prescribing simplicity and elegance for the sake of the user [100, 146, 187, 208]. These early insights by researchers about reducing the complexity of error messages were largely confirmed by the fact that error rates can be used as an indirect measure of cognitive load [6, 7]. These kinds of recommendations about simplicity and minimalism have continued in both anecdotal reports [67] and empirical studies [14, 21, 106, 153, 207]. However, recent empirical research has begun to incorporate cognitive science and CLT directly into work on programming error messages [9, 162, 163]. The most extensive work in this area provides three guidelines to reduce cognitive load in programmers receiving error messages to maximise working memory: place relevant information near the offending code, reduce redundancy so the user does not process the same information twice, and use multiple modalities to provide feedback [91].

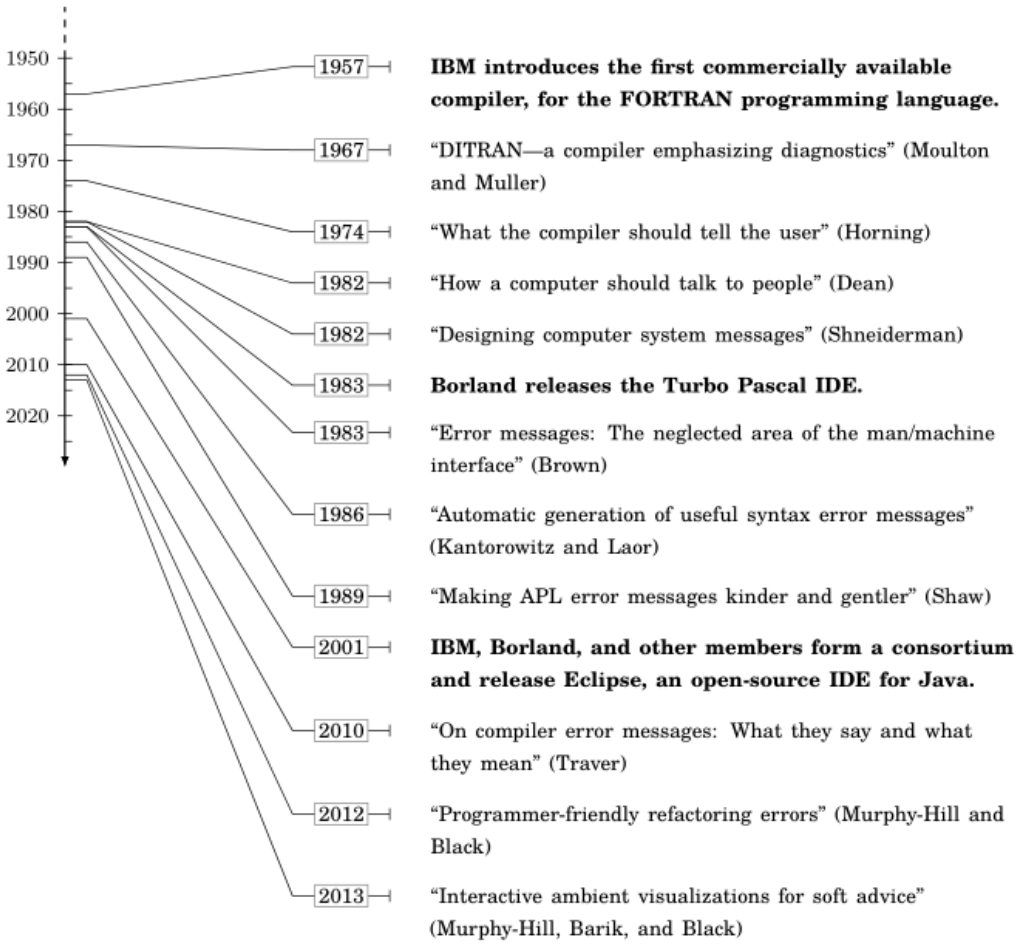

Figure 5: History of design guidelines for error messages in program analysis tools. In tandem with design guidelines, significant shifts in industry tools are indicated in bold. Key: Moulton & Muller [146]; Horning [90]; Dean [47]; Shneiderman [187]; Brown [36]; Kantorowitz & Laor [100]; Shaw [186]; Traver [202]; Murphy-Hill & Black [150]; Murphy-Hill, Barik, & Black [149]. To bring this timeline up to the current date, we would add [9]. Image and caption ©Titus Barik [9], used with permission.

8.3 Provide Context to the Error

The context of a programming error refers to information about the program code that is relevant to the error, and that will help make understanding and addressing the error easier (through an informed, accurate message). For example, the context can include the location in the code where the error occurs as well as information such as symbols, identifiers, literals, and types, involved in the error [69, 90, 123, 201, 215]. For run-time errors, the contextual information extends to the state of the memory during execution, including variable values and stack traces [90, 124]. In early text-based computer interfaces, the editor and compiler were typically separate programs (both in design and use) so providing locational context consisted of printing the code around the error that occurred [36, 100, 124, 186]. With the introduction of the integrated development environment, the error message can be displayed simultaneously with source code in the editing environment [67, 69]. In more modern development environments, the editing interface can graphically highlight the error location or place error messages beside offending code [91, 150, 212, 214].

Despite context being important, at least some programming error messages, in languages such as Java, can imply more than one context which could seem like providing the *wrong* context, as noted by Kölling & McCall [138, p2] (also mentioned in Section 1.1):

- A single error may, in different context, produce different diagnostic messages, and
- The same diagnostic message may be produced by entirely different and distinct errors.

Barik [9] recommends that error messages appear in the text editor, the primary construct through which the programmer interacts with code (see Figure 7). Becker [18] advocates for the original and enhanced error messages (see Section 7) to be presented side-by-side so that students can become accustomed to the cryptic

Table 3: Generalised guidelines extracted from the corpus compared to those presented in Horning [90], Shneiderman [187], Traver [202], and Barik [9].

Corpus	Horning (1976)	Shneiderman (1982)	Traver (2010)	Barik (2018)
Increase Readability	Concise yet distinctive, User directed	Comprehensible	Clarity and brevity, Programmer language	Implement rational reconstructions for humans, not tools
Reduce Cognitive Load	Source oriented, Readable, Specific	Brief	Consistency, Specificity	
Provide Context	Localize the problem, Visible pointer	Specific	Locality	Use code as the medium through which to situate error messages
Use a Positive Tone	Restrained and polite	Positive, Constructive, Emphasize user control over the system	Positive tone, Nonanthropomorphism	
Show Examples				
Show Solutions or Hints	Suggest corrections		Constructive guidance	Distinguish fixes from explanations
Allow Dynamic Interaction			Extensible help	Give developers autonomy over error message presentation
Provide Scaffolding				
Use Logical Argumentation				Present rational reconstructions as coherent narratives of causes to symptoms
Report Errors at the Right Time				
Not mentioned in corpus	Standard format, Complete		Visual design, Context-insensitivity	

unenhanced error messages. However, there is conflicting evidence of the benefits of graphically highlighting error messages (as there is for syntax highlighting [23, 179]). Barik *et al.* [11] demonstrate that diagrammatic annotations of source code with highlights, descriptions, and arrows can help developers comprehend programming error messages. Nienaltowski *et al.* [153] found that highlighting the error location in code helped students fix errors more quickly but did not increase novice programmers' ability to correct errors. Regardless, there is agreement on the evidence for the importance of the accuracy of error location reports. Traver [202] notes that providing the user with an incorrect error location makes the message more confusing for the programmer. Similarly, Wrenn & Krishnamurthi [212] report that students found highlighting of the error location in code to be helpful when it is correct, but frustrating when it is incorrect.

8.4 Use a Positive Tone

It is a tenet of human computer interaction that a computer's communication is akin to human communication. Therefore, how a computer communicates, or its tone, is just as important as what it communicates. As Shneiderman points out, this is a problem "when novices encounter violent messages ..., vague phrases ... , or obscure codes ... , they are understandably shaken, confused, dismayed, and discouraged from continuing" [187, p610].

There is a universal agreement in the literature that error messages should have a positive tone [64, 91, 187, 202]. This is described in a variety of ways including 'polite' [37, 89, 90], 'restrained' [90], 'friendly' [47, 148, 209], and 'encouraging' [91]. Horning [90] even claims that the computer should seem subservient to the programmer when reporting error messages. Other positive tone-based principles focus on avoiding words with a negative valence, like 'incorrect', 'illegal', and 'invalid' [64, 187, 202]. The positive-tone design principle also appears in guidelines to avoid in crafting error messages including avoiding placing fault or blame, scolding, or condemning the user [37, 47, 64, 93, 145, 187, 202, 209]. Buxton & Trennor argue that "Programmers need to see error messages not just as the functional reporting of errors, but as a means of communicating with users to increase their efficiency" [37, p197] and saw the friendliness of error messages as being an important part of this communication.

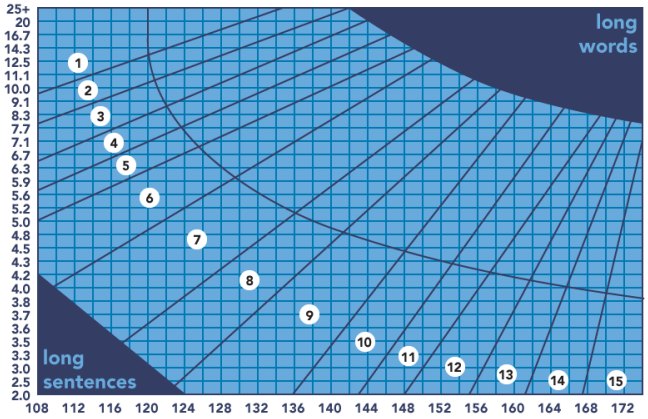

Figure 6: Fry readability graph. The reading difficulty level (white circles) is determined by the average number of sentences (y-axis) and syllables (x-axis) per 100 words. When plotted onto a specific graph the intersection of the average number of sentences and syllables determines the reading level of the content. Image: © user 'J' at English Wikipedia.

```
-- [E002] Syntax Warning: test/try.scala ---------
3 | try {
  | ^
  | A try without catch or finally is equivalent
  | to putting its body in a block; no exceptions
  | are handled.
4 |   foo()
5 | }
```

Figure 7: Programming error message generated by the Dotty compiler (Dotty is a superset of Scala) [147]. Notice the error appears inline with the code and each error is visually separated via code locality.

```
 1 program figure_1;
 2 var
 3     Grade = integer; <Students score>
-------------^   Syntax Error
You must use a ':' when declaring a variable.
For example: 'Age : Integer;'

  4
  5 begin <Start of Main Program>
  6     write <'Enter grade : '>;
  7     readln <grade>;
  8     if grade > 90 then
  9        writeln <'An "A"!'>;
------------------------------^   Syntax Error
This is a BAAAAAAAD SemiColon !!!!!
Hasn't your instructor told you 1,000 times
to NEVER put a SemiColon before an ELSE ???!!?
That SemiColon is ENDING the 'if' statement and
causing the 'else' to be the start of a new statement.

 10     else
 11        writeln <'Try Harder!'>;
 12 end. <End of Main Program>
```

Figure 8: Example of feedback messages in the CAP system described by Schorsch [182].

These suggestions lead to the following general programming error message guideline: use a positive tone. Under debate, and worthy of further study into efficacy, are the tone-based design principles of anthropomorphisation and humour. In 1995, Schorsch wrote the tool CAP, which used humour in its error messages [182]. These messages were sarcastic, made light of the error by poking fun at the user, and often blamed the user as well (see Figure 8). CAP's use of humour is contrary to a growing body of evidence on how novices learn, may contribute to learners feeling like they don't "get it," and is against the tenets of human computer interaction. Flowers reported on enhanced error messages in Gauntlet using humour to create messages that are more fun, and they anecdotally report that students respond positively [72]. However, Isa *et al.* [93] & Wexelblat [208] argue against using humour in error messages because the humour of a message diminishes with subsequent exposure and the inclusion of humour is antithetical to creating brief and informative messages [13]. Humorous programming error messages can also be misinterpreted. Sarcasm, as per Schorsch [182] for example, could be perceived as blaming.

Lee & Ko [113] demonstrate effective use of anthropomorphised error feedback. They argue that it succeeds because novice programmers see programming tools as cold and judgemental and that by personifying the tool programmers can attribute failure to the computer instead of themselves. Shneiderman [188], however, argues that personifying the computer may mistakenly give users the impression that the computer is sentient or that it can think. This is most likely more important for younger learners or those with limited prior exposure to technology – two groups that are set to become more numerous in programming education in the near future. Anthropomorphising the computer may then lead a programmer to develop an incorrect mental model for how the computer works, a potential problem for writing correct programs.

8.5 Show Examples of Similar Errors

A novice programmer will most likely spend many more hours engaged with the programming environment than they will with an educator. The environment thus becomes a proxy for the educator, and the rate at which a student is able to correct their errors and learn the new programming language will be directly influenced by the quality and amount of relevant information provided as feedback by that environment.

At the time the environment detects an error in a programmer's code, it has access to (possibly a lot of) contextual information about the error and the context in which it is found. While it is important that an error is correctly and clearly reported, there appears to be no consensus as to how much of this additional contextual information the compiler should provide. As identified, brevity offers many advantages, but providing complete information is likely to improve the student's understanding of the error and reduce the likelihood of repeating the mistake. Central to this issue is the very role of the environment – whether it is to assist the programmer in correcting errors or more simply to just inform the programmer of the reasons errors occurred.

In the same manner that programming language textbooks introduce concepts by example, the environment/compiler/interpreter has the opportunity to provide a more detailed explanation of an

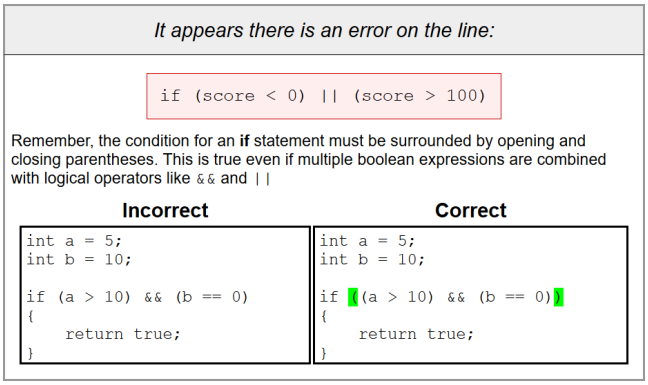

Figure 9: Programming error message generated by the CodeWrite system described by Denny *et al.* in [50]. The error message has been enhanced with an example of a similar error and its corresponding resolution. Displaying code examples side by side has been used in other programming tools such as the HelpMeOut system of Hartmann *et al.* [83].

error with an example. If an error is detected with a datatype or variable definition, or with the use of keywords for flow-control, the compiler can provide a relevant example of how the language's features should be used. While the example is guaranteed to be correct, this approach is pedagogically questionable in value as there is no guarantee that the programmer can relate the generic example to their erroneous code or has the knowledge to make sense of the example(s) provided.

Showing example code as part of the environment feedback was used by Denny *et al.* in their evaluation of an error message enhancement tool [50]. Upon detecting the type of error made by the student, the tool showed a similar incorrect example alongside an example showing the correct code, both accompanied by an explanation (see Figure 9). Example code was also displayed in the HelpMeOut system by Hartmann *et al.* [83]. HelpMeOut tracked code states from those with errors to those that were resolved, and provided these to users at compile time and run time.

Providing examples of similar errors is one of the more empirically validated guidelines. Multiple studies show that students perceive examples as helpful [154, 199, 207]. Other studies show that including examples yields significant benefits to problem completion and program understanding [110, 159].

However, Prather *et al.* conducted an empirical study on enhancing error messages, and one of the enhancements was showing example code [163]. They noted that showing novices example code only tended to confuse them. In their study, novices thought that the example code was *their* code and spent time looking for the example code in their code files. More work is necessary to validate this guideline as it currently has conflicting empirical evidence.

8.6 Show Solutions or Hints

Several authors assert that software tools should not merely report errors, but offer suggested solutions as to how they can be corrected. Such solutions need to be considered as *suggestions* and not as definitive advice, as the software tool cannot know the actual intent of the programmer (see Figure 10). The boundary between examples (discussed in Subsection 8.5) and solutions or hints is also not clearly defined and the user would need to know exactly which of these the provided information is.

Horning advocated for suggesting corrections in 1976 [90]. By 1986, Kantorowitz & Laor [100, p628] suggested that "A message that proposes how to correct an encountered error is most useful, but is only produced when there is a high degree of certainty for its correctness". Since that time, many other authors have advocated for providing solutions [89, 202] while others have warned against "leading a user down the wrong path..." [133, p3]. Some studies have shown solutions to be effective when used in high certainty situations [159, 174]. However, these studies mix solutions with other error message enhancements, so it is impossible to distinguish the impact of solutions on their own.

Kummerfeld & Kay [110] describe a software tool that suggests solutions to correct identified errors. A few lines of the original source code containing the error are displayed, with the identified error appearing in red text. A single suggested correction for the error appears alongside using vertical alignment and green text to identify the error and its corresponding suggested solution. They found that this guidance allowed novice programmers to repair errors approximately as fast as their more experienced counterparts.

Recent work by Thiselton & Treude [199] reports on a novel technique to augment identified Python error messages with formal Python documentation and crowd-sourced solutions to problems posted on the internet forum, Stack Overflow. Stack Overflow is employed by novice and experienced programmers alike. Questions about the meaning of programming error messages are frequently posted, and readers vote on the validity and helpfulness of the replies. The most popular responses percolate to the top of the list of replies. The forum offers extensive indexing and searching features, and esoteric questions and solutions, including just error numbers or hexadecimal memory addresses, can often be located.

Thiselton & Treude developed two software tools, one to augment errors with formal documentation providing *examples* of the correct use of a Python feature, and one providing suggested *solutions* to identify errors using highly ranked responses on Stack Overflow [199]. This makes clear the border between examples and suggested solutions we discussed at the beginning of this subsection. Evaluation of these tools was undertaken by 16 participants who encountered 115 Python syntax errors. The majority of participants agreed that summaries of responses from Stack Overflow, offering concrete suggestions for fixes and example code, were more helpful than the official Python documentation. When asked about the quality of the help providing details from formal Python documentation, one participant said *It's a bit too long*, while another stated that the assistance was too generic and *did not tell me what I should be doing*. The relatively low number of participants and errors provides opportunity for more robust results in the future.

8.7 Allow Dynamic Interaction

Several authors have advocated that programming language systems should not merely report their error messages, but engage the programmer with some level of dynamic interaction to elicit and provide more detail about an error. A common theme is that

```
Explanation
===========
A try expression should be followed by some
mechanism to handle any exceptions thrown.
Typically a catch expression follows the try and
pattern matches on any expected exceptions. For
example:

import scala.util.control.NonFatal

try {
  foo()
} catch {
  case NonFatal(e) => ???
}

It is also possible to follow a try immediately by
a finally - letting the exception propagate - but
still allowing for some clean up in finally:

try {
  foo()
} finally {
  //perform your cleanup here!
}

It is recommended to use the NonFatal extractor to
catch all exceptions as it correctly handles
transfer functions like return.
```

Figure 10: Programming error message generated by the Dotty compiler (Dotty is a superset of Scala) [147]. This is the same code that generated the error shown in Figure 7, but with the *-explain* flag passed to the compiler it provides more information and suggestions on how to fix the code.

program development environments should not provide a "one-shot" approach to error reporting, but should perform their role with programmers' histories of erroneous actions, and a possible estimation of their abilities. The success of such systems requires repeated use by students, perhaps in closed laboratories uniquely providing the augmented programming environment, or by using a cloud-based environment through which every interaction and outcome can be tracked.

Brown [36] summarises systems such as the Cornell Program Synthesizer, which tightly integrates editing with error analysis, and the "do what I mean" feature of Interlisp, which presents the user with a suggestion to correct an error. Brown suggests that the problem of poor messages may not be with the messages themselves; instead, he advocates for eliminating the possibility of the underlying error ever occurring. Similarly, McGiver & Conway [142] identify seven ways in which introductory programming languages hinder their own teaching, and call for error reporting mechanisms that ideally provide multiple levels of detail in error messages through a "tell-me-more" option. In contrast, Coull & Duncan [42] argue that a programming support tool for novice programmers should progressively *reduce* the use of enhanced compiler error messages as teaching progresses, so that students are gradually exposed to a compiler's true error messages.

Traver [202] discusses error messages from the perspective of human computer interaction, and states that the interface provided by poor error messages introduces significant barriers to students. He advocates for extensible help and environments where the knowledge of the programmer is taken into account. He suggests that compiler systems employing affective techniques, featuring emotional aspects, could help a desperate programmer by reporting recurring errors in different ways. The idea of 'empathetic' IDEs was briefly revisited in 2017 [46].

Suhailan *et al.* [194] describe the opportunities that social networks and logged social discussion to provide for automated augmented error reporting that may increase students' motivation. Watson *et al.* [207] report on their tool *BlueFix* which provides increasingly detailed error messages to students as they make multiple unsuccessful attempts at correcting an error. After three attempts, the student is presented with notes and crowd-sourced fixes demonstrating how other students have resolved similar problems. More recently, Thiselton & Treude [199] employed a similar approach of reporting errors augmented with crowd-sourced fixes using contributions to the Stack Overflow website. The two former studies were discussed in more detail in Section 7.

8.8 Provide Scaffolding for User

The concept of cognitive scaffolding is to provide a structure around which novices can place and build their knowledge [65, 131]. Even though it only emerged in the 1990s in computing education and has been rarely directly discussed in the literature, researchers have been proposing related ideas since the 1960s [208]. Many researchers discuss matching student expectations to what is presented to them in error messages, providing explanations of why they are receiving the error message, and providing support to the programmer as they learn key constructs and relationships. [72, 74, 214]. As previously discussed, Coull & Duncan proposed providing robust support to a student as they learn programming and then progressively reducing that support as the semester draws to a close [42]. Also in 2011, Marceau *et al.* argued that error messages should help students make the connection between the text of the message and the offending code, between terminology and code fragments, and between message and solution without explicitly providing a solution [132, 133]. These suggestions continue the long trajectory of arguing that error messages should provide scaffolding to the user without explicitly engaging cognitive learning theories [8, 10, 11, 163, 201, 207]. Recent work by Loksa *et al.* [121], Prather *et al.* [160–162] and Denny *et al.* [54] has begun engaging literature on metacognition in novice programming, of which one core element is to provide scaffolding through the entire learning process including when dealing with error messages. Very little research has been conducted on scaffolding the programmer while reading error messages that explicitly engages learning theory and remains one area that is wide open for further work.

Contrary to this idea is the very recent work of McCall & Kölling who argue that it is impossible to know what kind of misconception by the novice programmer caused the error and it is therefore imperative that the error message returned to the user remain imprecise and broad [140]. This appears to be the best argument against scaffolding and instead for concise, general error message

reporting. McCall & Kölling call into question some of the basic assumptions made by previous researchers and further empirical validation of scaffolding in programming error messages seems warranted.

8.9 Use Logical Argumentation

A very recent development in the study of programming error messages is a focus on using correct logical argumentation [9, 10]. Barik argues that error messages should be viewed through the lens of 'rational reconstructions' making concrete claims and providing sufficient warrants for those claims in order to be properly understood by the reader. Although Barik is the only author to propose this in the 50+ year history of the study of programming error messages, it is an important new development and represents an area needing replication and extension to continue this thread of research.

8.10 Report Errors at the Right Time

Finally, another recent development in the research on programming error messages is the idea that they should be presented to the user at the right time. Writers from the 1960s and 1970s often brought up this issue because one needed to see as many errors as possible when running a FORTRAN program took hours and not minutes or seconds (or less). It would have been impossible to debug a program in a reasonable amount of time if only one error was presented to the user at a time. This concern faded during the 1980s and 1990s as personal computers became ubiquitous. However, as computers and development environments increased in power, this once again became a concern. In the 2000s, researchers discussed the question of when to provide certain types of error messages (e.g. syntax or semantic) to the user [69, 72]. Recent work argues that programming error generation should use static analysis tools to show users error messages as soon as possible [159, 178]. This often appears in modern development tools as the "red squiggle" – a red underline that appears below problematic pieces or lines of code – but can take other forms as well. This trend follows Barik's idea that guidelines in the literature tend to run in tandem with shifts in industry tools [9].

9 CONCLUSIONS

We conclude by presenting the essential insights from each of Sections 4-8 before discussing future research directions.

9.1 Insights from Literature Search

One of the primary contributions of this work is the development of a comprehensive corpus of the literature on programming error messages. This corpus consists of 307 papers resulting from the combination of two sources. The first, a manually curated corpus crafted over 7 years, including articles that are unpublished, out-of-print, or not available online. The second, generated from a quasi-systematic, repeatable search of the literature. The final corpus bibtex file is available online[17] and remains a work in progress, being updated on an ongoing basis. no Articles in the corpus date back to the 1960s, yet most are recent with more than half being published in 2011 or later, reflecting a strong interest in programming error messages. We find that many of the articles are related to the idea of message enhancement, which is a topic of interest for both the education, human computer interaction, and programming language communities.

[17] https://iticse19-wg10.github.io/

9.2 Educational Insights

One of our most striking observations was that there was relatively little literature on the effect of programming error messages on students and their learning. Only 43 (14%) of the papers in our corpus were tagged with 'pedagogy'. Perhaps more telling is that 37 of these were collected manually as part of our 'original' corpus (which was curated in a much more ad hoc manner) compared to our quasi-systematic search which uncovered 19 (13 of these appeared in both sources).

9.3 Technical Insights

Implementing effective error messages in a compiler or interpreter is subject to many technical challenges, both theoretical and practical. Detecting erroneous program behaviour is an undecidable problem, so an error-detection system must choose between rejecting programs that would run correctly (false positives), or failing to detect some errors (false negatives). Then if an error is detected, it may be detected far from its source, making it challenging for a compiler to identify the actual location in code that generated the error. Programming systems that make use of code transformations also pose a problem; if an error is detected in the transformed code, it needs to be mapped back to the original, pre-transformed code, but the information to do so may no longer be available. In general, improving error detection and reporting is an engineering challenge that requires dedicated resources, and most compiler-development teams prioritise adding features over improving error diagnostics. Finally, live-compilation environments must deal with incomplete programs, while compilers are designed to work with the full source code of a program or module.

There is active research to improve programming error messages in both the programming language and computing education communities. We believe there is potential for both communities, along with experts in human computer interaction and machine learning, to collaborate on exploring new techniques and incorporating them into production systems, improving the programming experience for novice and expert programmers alike.

9.4 Insights from Enhancement

The area of programming error message enhancement, based on our corpus, spans the entire history of programming error messages in the academic literature. The efforts made in this area, apparently quite sporadic and sparse, are considerable when one looks at more than 50 years of research. There are two things that stand out in this body of work when taken as a whole:

- Efforts to improve programming error messages are significant, and this is perhaps the largest conspicuous piece of evidence for the fact that in general, programming error messages are largely ineffective; and
- There has been little progress in more than 50 years. Despite new languages, data collection possibilities, and increased

interest of late, there are remarkable similarities between the motivating claims and research outputs for studies that span half of a century, and little consensus on what the best way forward is today.

Although here we discuss programming error messages specifically, our conclusions are reminiscent of observations by Jadud in 2006 when discussing programming environments more generally: "Despite massive increases in the computational power casually available to instructors and students, the tools used to write and compile programs have changed minimally in the last fifty years" [98, p7].

It is most likely that significant progress will not be made without larger-scale, concerted efforts and in particular, replication studies. Most importantly these efforts will need to span multiple academic communities including Computing Education, Programming Languages, and Human Computer Interaction. This would be a very interesting and possibly quite impactful area for collaboration between SIGCSE, SIGPLAN, and SIGCHI.

It is possible that other advances will in effect solve the issues that enhancement efforts have been trying to solve for decades. To name a few there are: automatic hint-generating systems that show promise; advances in machine learning techniques that identify/correct errors often completely bypassing the traditional text-based messages; advances in programming environment technology (background compilation, highlighting, etc.) that sometimes alleviate (but also sometimes add to) the problem; and data mining, crowd sourcing and other data-driven approaches to doing what programming error messages are supposed to do – help resolve errors in code effectively.

If anything, the history of compiler error enhancement reaffirms one thing – humans are still best at some tasks, and fixing programming errors by interpreting and acting on programming error messages seems to be one of those tasks even with little or poor help. It would be good however, if the systems we work with made things easier on the humans. Generating more effective programming error messages would be a big step in that direction.

9.5 Insights from Guidelines

Guidelines for authoring programming error messages are scattered throughout the literature with sparse agreement on what constitutes an effective programming error message. Dozens of papers appearing before the year 2000 discuss issues with programming error messages, but either do not provide guidelines or list them anecdotally. This trend has begun to reverse in recent years, with many publications providing empirical evidence for guidelines, but this evidence is often weak, not robust, and not repeatable. The field is also continuing to evolve as technologies change. Guidelines relevant to the limitations of punch cards or terminals are no longer applicable in the age of personal computing. Some 13 years ago Jadud had already declared that modern development environments had taken the "rapid compilation cycle" to its "natural limit" [98, p8].

We continue to see theories new to computer science education applied to programming error messages, such as Cognitive Load Theory, advancing the state-of-the-art in exciting ways. This working group created a compendium of all of the recommendations in one usable format (see Tables 2 and 3). These tables provide the research community with a master list of effective programming error message design. We collected and organised the guidelines from the literature into useful groupings; the first time these guidelines have been gathered in one place. There is, however, still a need for empirical validation of our list.

9.6 Call for Research

9.6.1 Readability. As discussed in Section 8.1, there is agreement among authors that **readability** is an important aspect of a good programming error message. Readability of natural language has been defined in various ways and is well-studied [44, 58, 73, 82, 136]. As far as we know these definitions have not been specifically applied to programming error messages, and metrics for evaluating readability of error messages still need to be formally defined. The impact of readability on programmers remains to be evaluated in isolation, independently from other improvements to error messages. It is arguable that enhancement efforts will not be as impactful as they could be, if basics such as readability are not well-researched.

9.6.2 Guidelines. Many guidelines for creating error messages exist and some appear to be widely accepted. However there is very little empirical evidence that supports either a particular guideline or a related pedagogic practice. Individual guidelines should be examined and then robustly tested to determine their effectiveness.

9.6.3 Message Identifiers. Historically, many compilers have associated unique identifiers with each error message, allowing the error to be looked up in a paper manual or on a website. Microsoft's Visual Studio compiler includes message identifiers, and its support documentation provides more detail for each identifier, including an explanation of the error, what could cause it, and suggestions for how to fix it. Languages include C/C++[18] and C#[19]. An open area of research is whether or not such identifiers and external documentation facilitate programmer understanding of error messages and reduce the time it takes to correct errors.

9.7 Moving Ahead

In conclusion the following observations are abundantly clear upon completing this work:

(A) To-date the literature on programming error messages is sparse and scattered, although interest has increased dramatically in the last 5-10 years.
(B) Programming error messages are problematic, regardless of language, particularly for students. They have been for over 50 years, and progress has been slow. They are likely to remain problematic for some time.
(C) Programming error messages are pedagogically important, particularly in their roles as feedback agents. This is unlikely to change for some time.
(D) Programming error messages are technically difficult to perfect ab initio. This is unlikely to be resolved soon. This has resulted in several different approaches to engineering them

[18] https://docs.microsoft.com/en-us/cpp/error-messages/compiler-errors-1/c-cpp-build-errors?view=vs-2019
[19] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/

post hoc or eliminating them through other means. These have met limited success.

(E) While there have been many guidelines for programming error message design proposed and implemented, no coherent picture has emerged to answer even simple questions such as: *What does a good programming error message look like?*

With a view to moving forward, we can state the following requirements for progress:

(a) In addition to research leveraging the latest developments in programming language design, and technologies such as machine learning, research at basic human computer interaction levels – such as how to measure the readability of programming error messages – is needed.

(b) More attention and effort is required so that research in this area does not continue the way it has for more than half of a century – largely in isolated groups. Only through collaborative efforts such as this, do we have the best chance to come to consensus on the best way forward.

The contributions of this working group are:

(1) The development of a comprehensive corpus of the literature on programming error messages.

(2) A compendium of programming error message design guidelines in one usable format.

(3) A bibtex bibliography of our combined corpus.[20]

We believe that contributions (1) and (2) will help the community move towards meeting requirements (a) and (b) with a view to changing (A) - (E). As our final conclusion we would like to state:

> Programming error messages are important but problematic, and have been for over half of a century. Without a more coordinated effort this is unlikely to change. Currently there is not even an agreed way to measure the effectiveness of programming error messages, effectively.

9.8 Afterword

If you've read this far, you know that we consider programming error messages to be serious, as they have genuine consequences, particularly for learners. Nonetheless it is important to look on the bright side sometimes, as shown in Figure 11.

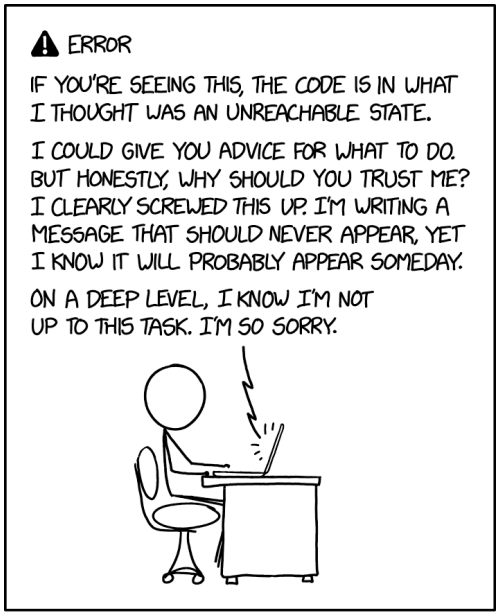

Figure 11: To err is human; to generate effective text-based programming error messages, divine. Image: From xkcd.com/2200/ ©Randall Munroe xkcd.com/license.html

ACKNOWLEDGMENTS

We would like to thank the following people for their help: Catherine Mooney for the initial scripting for our bibtex and tags and for her helpful comments; Heidi Nobles for her help with academic archaeology; Titus Barik for helpful suggestions, discussions, and permission to use his figure; and Amanpreet Kapoor, the ITiCSE doctoral consortium student who visited our working group and had excellent insight.

[20] https://iticse19-wg10.github.io/

REFERENCES

[1] Alireza Ahadi, Arto Hellas, Petri Ihantola, Ari Korhonen, and Andrew Petersen. 2016. Replication in Computing Education Research: Researcher Attitudes and Experiences. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/2999541.2999554

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An Analysis of Patterns of Debugging Among Novice Computer Science Students. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*. ACM, New York, NY, USA, 84–88. https://doi.org/10.1145/1067445.1067472

[3] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation Error Repair: For the Student Programs, from the Student Programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '18)*. ACM, New York, NY, USA, 78–87. https://doi.org/10.1145/3183377.3183383

[4] Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. 2019. Targeted Example Generation for Compilation Errors. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE '19)*. ACM, New York, NY, USA, 12.

[5] Amjad Altadmri, Michael Kolling, and Neil C. C. Brown. 2016. The Cost of Syntax and How to Avoid It: Text versus Frame-Based Editing. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC '16)*. IEEE, 748–753. https://doi.org/10.1109/COMPSAC.2016.204

[6] Paul Ayres and John Sweller. 1990. Locus of Difficulty in Multistage Mathematics Problems. *The American Journal of Psychology* 103, 2 (1990), 167–193. http://www.jstor.org/stable/1423141

[7] Paul L. Ayres. 2001. Systematic Mathematical Errors and Cognitive Load. *Contemporary Educational Psychology* 26, 2 (2001), 227 – 248. https://doi.org/10.1006/ceps.2000.1051

[8] Titus Barik. 2016. How Should Static Analysis Tools Explain Anomalies to Developers? A Communication Theory of Computationally Supporting Developer Self-Explanations for Static Analysis Anomalies. (2016). http://static.barik.net/barik/proposal/barik_proposal_approved.pdf

[9] Titus Barik. 2018. *Error Messages as Rational Reconstructions.* Ph.D. Dissertation. North Carolina State University, Raleigh. https://repository.lib.ncsu.edu/handle/1840.20/35439

[10] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 633–643. https://doi.org/10.1145/3236024.3236040

[11] Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill. 2014. How Developers Visualize Compiler Messages: A Foundational Approach to Notification Construction. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 87–96. https://doi.org/10.1109/VISSOFT.2014.24

[12] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 575–585. https://doi.org/10.1109/ICSE.2017.59

[13] D. W. Barron. 1975. A Note on APL. *Comput. J.* 19, 1 (1975), 93. https://academic.oup.com/comjnl/article-pdf/19/1/93/1058172/190093.pdf

[14] Brett A. Becker. 2015. *An Exploration Of The Effects Of Enhanced Compiler Error Messages For Computer Programming Novices*. Masters Thesis. Dublin Institute of Technology. https://doi.org/10.13140/RG.2.2.26637.13288

[15] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[16] Brett A. Becker. 2016. A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. In *Proceedings of the 21st ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. ACM, New York, NY, USA, 296–301. https://doi.org/10.1145/2899415.2899463

[17] Brett A. Becker. 2019. Parlez-vous Java? Bonjour La Monde != Hello World: Barriers to Programming Language Acquisition for Non-Native English Speakers. In *Proceedings of the 30th Annual Conference of the Psychology of Programming Interest Group (PPIG '19)*. http://www.ppig.org/sites/ppig.org/files/2019-PPIG-30th-becker.pdf

[18] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective Compiler Error Message Enhancement for Novice Programming Students. *Computer Science Education* 26, 2-3 (2016), 148–175. https://doi.org/10.1080/08993408.2016.1225464

[19] Brett A. Becker, Kyle Goslin, and Graham Glanville. 2018. The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 640–645. https://doi.org/10.1145/3159450.3159461

[20] Brett A. Becker and Catherine Mooney. 2016. Categorizing Compiler Error Messages with Principal Component Analysis. In *Proceedings of the 12th China-Europe International Symposium on Software Engineering Education (CEISEE '16)*. Shenyang, China, 1–8. https://researchrepository.ucd.ie/handle/10197/7889

[21] Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. 2018. Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 634–639. https://doi.org/10.1145/3159450.3159453

[22] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 338–344. https://doi.org/10.1145/3287324.3287432

[23] T.R. Beelders and Jean-Pierre L. du Plessis. 2016. The Influence of Syntax Highlighting on Reading and Comprehending Code. *Journal of Eye Movement Research Beelders* 91, 1 (2016), 1–11. https://doi.org/10.16910/jemr.9.1.1

[24] Mordechai (Moti) Ben-Ari. 2007. Compile and Runtime Errors in Java. (2007). http://www.weizmann.ac.il/sci-tea/benari/sites/sci-tea.benari/files/uploads/softwareAndLearningMaterials/errors.pdf

[25] Joe Bergin, Achla Agarwal, and Krishna Agarwal. 2003. Some Deficiencies of C++ in Teaching CS1 and CS2. *ACM SIGPLAN Notices* 38, 6 (2003), 9–13. https://doi.org/10.1145/885638.885642

[26] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. In *2nd Indian Workshop on Machine Learning (IWML '16)*. https://www2.cse.iitk.ac.in/~iwml/2016/papers/iWML_2016_paper_5.pdf

[27] Michael W. Bigrigg, Russell Bortz, Shyamal Chandra, David Reed, Jared Sheehan, and Sara Smith. 2003. *An Evaluation of the Usefulness of Compiler Error Messages*. Technical Report. http://www.ices.cmu.edu/reports/040903.pdf

[28] Dennis Bouvier, Ellie Lovellette, John Matta, Bedour Alshaigy, Brett A. Becker, Michelle Craig, Jana Jackova, Robert McCartney, Kate Sanders, and Mark Zarb. 2016. Novice Programmers and the Problem Description Effect. In *Proceedings of the 2016 ITiCSE Working Group Reports (ITiCSE-WGR '16)*. ACM, New York, NY, USA, 103–118. https://doi.org/10.1145/3024906.3024912

[29] Bernd Braßel. 2004. Typehope: There is Hope for Your Type Errors. In *Int. Workshop on Implementation of Functional Languages*.

[30] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from Applying the Systematic Literature Review Process Within the Software Engineering Domain. *J. Syst. Softw.* 80, 4 (April 2007), 571–583. https://doi.org/10.1016/j.jss.2006.07.009

[31] Neil C. C. Brown and Amjad Altadmri. 2017. Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. *ACM Transactions on Computing Education* 17, 2, Article 7 (May 2017), 21 pages. https://doi.org/10.1145/2994154

[32] Neil C. C. Brown, Amjad Altadmri, Sue Sentance, and Michael Kölling. 2018. Blackbox, Five Years On. In *Proceedings of the 2018 ACM Conference on International Computing Education Research - ICER '18*. ACM, Espoo, Finland, 196–204. https://doi.org/10.1145/3230977.3230991

[33] Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 223–228. https://doi.org/10.1145/2538862.2538924

[34] Neil C. C. Brown and Greg Wilson. 2018. Ten Quick Tips For Teaching Programming. *PLoS Computational Biology* 14, 4 (Apr 2018), e1006023. https://doi.org/10.1371/journal.pcbi.1006023

[35] P. J. Brown. 1982. My System Gives Excellent Error Messages - Or Does It? *Software: Practice and Experience* 12, 1 (Jan 1982), 91–94. https://doi.org/10.1002/spe.4380120110

[36] P. J. Brown. 1983. Error Messages: The Neglected Area of the Man/Machine Interface. *Commun. ACM* 26, 4 (Apr 1983), 246–249. https://doi.org/10.1145/2163.358083

[37] Andrew Buxton and Lesley Trenner. 1987. An Experiment to Assess the Friendliness of Error Messages from Interactive Information Retrieval Systems. *Journal of Information Science* 13, 4 (Aug 1987), 197–209. https://doi.org/10.1177/016555158701300403

[38] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. 2015. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/2787622.2787710

[39] A. T. Chamillard and William C. Hobart, Jr. 1997. Transitioning to Ada in an Introductory Course for Non-majors. In *Proceedings of the Conference on TRI-Ada (TRI-Ada '97)*. ACM, New York, NY, USA, 37–40. https://doi.org/10.1145/269629.269634

[40] G. V. Cormack. 1989. An LR Substring Parser for Noncorrecting Syntax Error Recovery. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*. ACM, New York, NY, USA, 161–169. https://doi.org/10.1145/73141.74832

[41] Natalie J Coull. 2008. *SNOOPIE: Development of a Learning Support Tool for Novice Programmers within a Conceptual Framework*. Ph.D. Dissertation. University of St Andrews, St Andrews, Scotland. http://hdl.handle.net/10023/522

[42] Natalie J. Coull and Ishbel M.M. Duncan. 2011. Emergent Requirements for Supporting Introductory Programming. *Innovations in Teaching and Learning in Information and Computer Sciences (ITaLICS)* 10, 1 (2011), 78–85. https://doi.org/10.11120/ital.2011.10010078

[43] Paul Cress, Paul Dirksen, and Wesley J Graham. 1970. *FORTRAN IV With WATFOR and WATFIV*. Prentice-Hall, Englewood Cliffs, New Jersey.

[44] Edgar Dale and Jeanne S Chall. 1949. The Concept of Readability. *Elementary English* 26, 1 (1949), 19–26.

[45] E A Davis, M C Linn, and M Clancy. 1995. Learning to Use Parentheses and Quotes in LISP. *Computer Science Education* 6, 1 (1995), 15–31. https://doi.org/10.1080/0899340950060102

[46] Carla De Lira. 2017. Improving the Learning Experiences of First-Year Computer Science Students with Empathetic IDEs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 293–294. https://doi.org/10.1145/3105726.3105742

[47] Morris Dean. 1982. How a Computer Should Talk To People. *IBM Systems Journal* 21, 4 (1982), 424–453. https://doi.org/10.1147/sj.214.0424

[48] Paul Denny, Brett A. Becker, Michelle Craig, Greg Wilson, and Piotr Banaszkiewicz. 2019. Research This! Questions That Computing Educators Most Want Computing Education Researchers to Answer. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. ACM, New York, NY, USA, 259–267. https://doi.org/10.1145/3291279.3339402

[49] Paul Denny, Brian Hanks, and Beth Simon. 2010. PeerWise: Replication Study of a Student-collaborative Self-testing Web Service in a U.S. Setting. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. ACM, New York, NY, USA, 421–425. https://doi.org/10.1145/1734263.1734407

[50] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 19th Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, NY, USA, 273–278. https://doi.org/10.1145/2591708.2591748

[51] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All Syntax Errors Are Not Equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. ACM, New York, NY, USA, 75–80. https://doi.org/10.1145/2325296.2325318

[52] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. CodeWrite: Supporting Student-driven Practice of Java. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*. ACM, New York, NY, USA, 471–476. https://doi.org/10.1145/1953163.1953299

[53] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. ACM, New York, NY, USA, 208–212. https://doi.org/10.1145/1999747.1999807

[54] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research (Koli Calling '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3364510.3366170

[55] Gergely Dévai, Dániel Leskó, and Máté Tejfel. 2013. *The EDSL's Struggle for Their Sources*. In: Zsók V., Horváth Z., Csató L. (eds) Central European Functional Programming School. CEFP 2013. Lecture Notes in Computer Science, Vol. 8606. Springer, Cham. 300–335 pages. https://doi.org/10.1007/978-3-319-15940-9_7

[56] Tao Dong and Kandarp Khandwala. 2019. The Impact of "Cosmetic" Changes on the Usability of Error Messages. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–6. https://doi.org/10.1145/3290607.3312978

[57] Benedict du Boulay and Ian Matthew. 1984. Fatal Error in Pass Zero: How Not to Confuse Novices. *Behaviour and Information Technology* 3, 2 (1984), 109–118. https://doi.org/10.1080/01449298408901742

[58] William H DuBay. 2007. *Smart Language: Readers, Readability, and the Grading of Text*. ERIC.

[59] Thomas Dy and Ma. Mercedes Rodrigo. 2010. A Detector for Non-literal Java Errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 118–122. https://doi.org/10.1145/1930464.1930485

[60] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle and José Nelson Amaral. 2018. *Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors*. 311–322 pages.

[61] Marc Eisenstadt and Matthew W. Lewis. 2018. Errors in an Interactive Programming Environment: Causes and Cures. In *Novice Programming Environments*, Mark Eisenstadt, Mark T. Keane, and Tim Rajan (Eds.). Routledge, London, Chapter 5, 111–131.

[62] Nabil El Boustani and Jurriaan Hage. 2010. Corrective Hints for Type Incorrect Generic Java Programs. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '10)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/1706356.1706360

[63] Nabil El Boustani and Jurriaan Hage. 2011. Improving Type Error Messages for Generic Java. In *Higher-Order and Symbolic Computation*, Vol. 24. Savannah, GA, 3–39. https://doi.org/10.1007/s10990-011-9070-3

[64] U. Engelmann and H. P. Meinzer. 1985. Rules for the Design of End User Languages. In *Medical Informatics Europe 85*, F. H. Roger, P. Grönroos, R. Tervo-Pellikka, and R. O'Moore (Eds.). Springer, Berlin, Heidelberg, Helsinki, Finland, 240–245. https://doi.org/10.1007/978-3-642-93295-3_48

[65] Anneli Eteläpelto. 1993. Metacognition and the Expertise of Computer Program Comprehension. *Scandinavian Journal of Educational Research* 37, 3 (1993), 243–254. https://doi.org/10.1080/0031383930370305

[66] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common Logic Errors Made by Novice Programmers. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. ACM, New York, NY, USA, 83–89. https://doi.org/10.1145/3160489.3160493

[67] Georgios Evangelidis, Vassilios Dagdilelis, Maria Satratzemi, and Vassilios Efopoulos. 2001. X-compiler: Yet Another Integrated Novice Programming Environment. In *Proceedings IEEE International Conference on Advanced Learning Technologies*. IEEE Comput. Soc, 166–169. https://doi.org/10.1109/ICALT.2001.943890

[68] S. Fincher. 1999. What are we doing when we teach programming?. In *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011*, Vol. 1. 12A4/1–12A4/5 vol.1. https://doi.org/10.1109/FIE.1999.839268

[69] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming* 12, 2 (March 2002), 159–182. https://doi.org/10.1017/S0956796801004208

[70] Allan Fisher and Jane Margolis. 2002. Unlocking the Clubhouse: The Carnegie Mellon Experience. *SIGCSE Bull.* 34, 2 (June 2002), 79–83. https://doi.org/10.1145/543812.543836

[71] Julie Fisher. 1999. The Importance of User Message Text and Why Professional Writers Should Be Involved. *Australian Computer Journal* 31 (Nov 1999), 118–123.

[72] Thomas Flowers, Curtis Carver, and James Jackson. 2004. Empowering students and building confidence in novice programmers through gauntlet. In *34th ASEE/IEEE Annual Frontiers in Education, 2004. FIE 2004.* IEEE, Savannah, GA, USA, T3H10–13. https://doi.org/10.1109/fie.2004.1408551

[73] Edward B Fry. 2006. Readability: Reading Hall of Fame Book. Newark. *DE: International Reading Association* (2006).

[74] Richard Furuta and P. Michael Kemp. 1979. Experimental Evaluation of Programming Language Features: Implications for Introductory Programming Languages. In *Proceedings of the 10th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '79)*. ACM, New York, NY, USA, 18–21. https://doi.org/10.1145/800126.809544

[75] Susan L. Graham and Steven P. Rhodes. 1973. Practical Syntactic Error Recovery in Compilers. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. ACM, New York, NY, USA, 52–58. https://doi.org/10.1145/512927.512932

[76] David Gries. 1968. Use of Transition Matrices in Compiling. *Commun. ACM* 11, 1 (Jan. 1968), 26–34. https://doi.org/10.1145/362851.362872

[77] David Gries. 1974. What Should We Teach in an Introductory Programming Course?. In *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '74)*. ACM, New York, NY, USA, 81–89. https://doi.org/10.1145/800183.810447

[78] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, 1345–1351. http://dl.acm.org/citation.cfm?id=3298239.3298436

[79] Jurriaan Hage and Heeren Bastiaan. 2006. Heuristics for Type Error Discovery and Recovery. In *18th International Conference on Implementation and Application of Functional Languages (IFL '06)*. 199–216. https://link.springer.com/chapter/10.1007/978-3-540-74130-5_12

[80] Devon Harker. 2017. *Examining the Effects of Enhanced Compilers on Student Productivity*. Masters Thesis. University of Northern British Columbia. https://unbc.arcabc.ca/islandora/object/unbc%3A58897

[81] Jan Lee Harrington. 1984. *The Effect of Error Messages on Learning Computer Programming by Individuals Without Prior Programming Experience*. PhD Thesis. Drexel University.

[82] Theodore L Harris and Richard E Hodges. 1995. *The Literacy Dictionary: The Vocabulary of Reading and Writing*. ERIC.

[83] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1019–1028. https://doi.org/10.1145/1753326.1753478

[84] J. Hartz, Adam. 2012. *CAT-SOOP: A Tool for Automatic Collection and Assessment of Homework Exercises*. Master's thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/bitstream/handle/1721.1/77086/825763362-MIT.pdf?sequence=2

[85] Brian Harvey. 1982. Why Logo? *Byte* 7, 8 (Aug 1982), 163–195. http://cmkfutures.com/wp-content/uploads/2017/06/Why-Logo-by-Brian-Harvey.pdf

[86] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the Type Inference Process. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, New York, NY, USA, 3–13. https://doi.org/10.1145/944705.944707

[87] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for Learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/871895.871902

[88] Zef Hemel, Danny M. Groenewegen, Lennart C.L. Kats, and Eelco Visser. 2011. Static Consistency Checking of Web Applications with WebDSL. *Journal of Symbolic Computation* 46, 2 (2011), 150–182. https://doi.org/10.1016/j.jsc.2010.08.006

[89] Richard Hill. 2008. *Developing a Teaching Compiler for Students Learning the C Programming Language*. Bachelor of Science Dissertation. University of Bath.

[90] James J Horning. 1976. What the Compiler Should Tell the User. In *Compiler Construction: An Advanced Course*, G Goos and J Hartmanis (Eds.). Springer-Verlag, Berlin-Heidelberg, 525–548.

[91] C. D. Hundhausen, D. M. Olivares, and A. S. Carter. 2017. IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda. *ACM Transactions on Computing Education* 17, 3, Article 11 (Aug. 2017), 26 pages. https://doi.org/10.1145/3105759

[92] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE Working Group Reports (ITICSE-WGR '15)*. ACM, New York, NY, USA, 41–63. https://doi.org/10.1145/2858796.2858798

[93] Barbara S. Isa, James M. Boyle, Alan S. Neal, and Roger M. Simons. 1983. A Methodology for Objectively Evaluating Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '83)*. ACM, New York, NY, USA, 68–71. https://doi.org/10.1145/800045.801583

[94] ISO/IEC 14882:2011 2011. *Information Technology – Programming languages – C++*. Technical Report. https://www.iso.org/standard/50372.html

[95] ISO/IEC TS 19217:2015 2015. *Information Technology – Programming Languages – C++ Extensions for Concepts*. Technical Report. https://www.iso.org/standard/64031.html

[96] J. Jackson, M. Cobb, and C. Carver. 2005. Identifying Top Java Errors for Novice Programmers. In *35th Annual Frontiers in Education Conference (FIE '05)*. T4C–24 – T4C–27. https://doi.org/10.1109/fie.2005.1611967

[97] Matthew C Jadud. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (2005), 25–40. https://doi.org/10.1080/08993400500056530 arXiv:https://doi.org/10.1080/08993400500056530

[98] Matthew C. Jadud. 2006. *An Exploration of Novice Compilation Behaviour in BlueJ*. Ph.D. Dissertation. University of Kent at Canterbury. https://jadud.com/dl/pdf/jadud-dissertation.pdf

[99] Mathias Johan Johansen. 2015. *Errors and Misunderstandings Among Novice Programmers Assessing the Student Not the Program*. Masters Thesis. University of Oslo. https://www.duo.uio.no/handle/10852/49045

[100] Eliezer Kantorowitz and H. Laor. 1986. Automatic Generation of Useful Syntax Error Messages. *Software: Practice and Experience* 16, 7 (1986), 627–640.

[101] Ioannis Karvelas. 2019. Investigating Novice Programmers' Interaction with Programming Environments. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 336–337. https://doi.org/10.1145/3304221.3325596

[102] Caitlin Kelleher, Dennis Cosgrove, and David Culyba. 2002. Alice2: Programming Without Syntax Errors. *User Interface Software and Technology - UIST 2002* (2002), 3–4. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.4640&rep=rep1&type=pdf

[103] B. Kitchenham and S. Charters. 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering, version 2.3. (2007).

[104] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems Prior Research on Learning Barriers A Study of Visual Basic . NET 2003. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '04)*. 199–206. https://doi.org/10.1109/VLHCC.2004.47

[105] Tobias Kohn. 2017. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. PhD Thesis. ETH Zürich.

[106] Tobias Kohn. 2019. The Error Behind The Message: Finding the Cause of Error Messages in Python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 524–530. https://doi.org/10.1145/3287324.3287381

[107] Michael Kölling. 1999. *The Design of an Object-Oriented Environment and Language for Teaching*. Ph.D. Dissertation. University of Sydney. https://kar.kent.ac.uk/21868/1/the_design_of_an_object-oriented_kolling.pdf

[108] Michael Kölling. 2015. Lessons from the Design of Three Educational Programming Environments: Blue, BlueJ and Greenfoot. *International Journal of People-Oriented Programming* 4, 1 (2015), 5–32. https://doi.org/10.4018/ijpop.2015010102

[109] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ System and its Pedagogy. *Computer Science Education* 13, 4 (Dec 2003), 249–268. https://doi.org/10.1076/csed.13.4.249.17496

[110] Sarah K. Kummerfeld and Judy Kay. 2003. The Neglected Battle Fields of Syntax Errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20 (ACE '03)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 105–111. http://dl.acm.org/citation.cfm?id=858403.858416

[111] Thomas Kurtz. 1978. BASIC. *ACM SIGPLAN Notices - Special issue: History of programming languages conference* 13, 8 (1978), 103–118. https://doi.org/10.1145/960118.808376

[112] Nicolas Laurent. 2017. Red Shift: Procedural Shift-reduce Parsing (Vision Paper). In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE '17)*. ACM, New York, NY, USA, 38–42. https://doi.org/10.1145/3136014.3136036

[113] Michael J. Lee and Amy J. Ko. 2011. Personifying Programming Tool Feedback Improves Novice Programmers' Learning. In *Proceedings of the Seventh International Workshop on Computing Education Research (ICER '11)*. ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/2016911.2016934

[114] Oukseh Lee and Kwangkeun Yi. 1998. Proofs About a Folklore Let-polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 707–723. https://doi.org/10.1145/291891.291892

[115] Ronald Paul Leinius. 1970. *Error Detection and Recovery for Syntax Directed Compiler Systems*. Ph.D. Dissertation. AAI7024758.

[116] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-error Messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 425–434. https://doi.org/10.1145/1250734.1250783

[117] Stuart Lewis and Gaius Mulley. 1998. A Comparison Between Novice and Experienced Compiler Users in a Learning Environment. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education (ITiCSE '98)*. ACM, New York, NY, USA, 157–161. https://doi.org/10.1145/282991.283106

[118] William Lidwell, Kritina Holden, and Jill Butler. 2010. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*. Rockport Publishers.

[119] Derrell Lipman. 2014. *LearnCS! a Browser-Based Research Platform for CS1 and Studying the Role of Instruction of Debugging from Early in the Course*. Ph.D. Dissertation. University of Massachusetts Lowell.

[120] Charles R. Litecky and Gordon B. Davis. 1976. A Study of Errors, Error-proneness, and Error Diagnosis in Cobol. *Commun. ACM* 19, 1 (1976), 33–38. https://doi.org/10.1145/359970.359991

[121] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252

[122] Glenn R Luecke, James Coyle, James Hoekstra, Marina Kraeva, and Ying Xu. 2009. The Importance of Run-time Error Detection. In *Tools for High Performance Computing 2009*. 145–155. https://doi.org/10.1007/978-3-642-11261-4

[123] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Elizabeth Kleiman, and Olga Weiss. 2009. Evaluating Error Detection Capabilities of UPC Run-time Systems. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models (PGAS '09)*. ACM, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/1809961.1809971

[124] Harri Luoma, Essi Lahtinen, and Hannu-Matti Järvinen. 2007. CLIP, a Command Line Interpreter for a Subset of C++. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88 (Koli Calling '07)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 199–202. http://dl.acm.org/citation.cfm?id=2449323.2449351

[125] Andrew Luxton-Reilly. 2016. Learning to Program is Easy. In *Proceedings of the 21st ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. ACM, New York, NY, USA, 284–289. https://doi.org/10.1145/2899415.2899432

[126] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Simon, and Jacqueline Whalley. 2017. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17)*. ACM, New York, NY, USA, 47–69. https://doi.org/10.1145/3174781.3174784

[127] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '18)*. ACM, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[128] Celeste S. Magers. 1983. An Experimental Evaluation of On-line HELP for Non-programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '83)*. ACM, New York, NY, USA, 277–281. https://doi.org/10.1145/800045.801626

[129] Qusay H. Mahmoud, Wlodek Dobosiewicz, and David Swayne. 2004. Redesigning Introductory Computer Programming with HTML, JavaScript, and Java. *SIGCSE Bull.* 36, 1 (March 2004), 120–124. https://doi.org/10.1145/1028174.971344

[130] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[131] Murali Mani and Quamrul Mazumder. 2013. Incorporating Metacognition into Learning. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 53–58. https://doi.org/10.1145/2445196.2445218

[132] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*. ACM, New York, NY, USA, 499–504. https://doi.org/10.1145/1953163.1953308

[133] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

[134] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The Impact of Adding Textual Explanations to Next-step Hints in a Novice Programming Environment. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 520–526. https://doi.org/10.1145/3304221.3319759

[135] Richard E. Mayer. 2004. Teaching of Subject Matter. *Annual Review of Psychology* 55, 1 (Feb 2004), 715–744. https://doi.org/10.1146/annurev.psych.55.082602.133124

[136] G Harry Mc Laughlin. 1969. SMOG Grading – A New Readability Formula. *Journal of reading* 12, 8 (1969), 639–646.

[137] Bruce J. McAdam. 1998. *On the Unification of Substitutions in Type Inference.* Technical Report. 1–23 pages. http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-384/ECS-LFCS-98-384.pdf

[138] Davin Mccall. 2016. *Novice Programmer Errors-Analysis and Diagnostics.* Ph.D. Dissertation. The University of Kent. https://kar.kent.ac.uk/id/eprint/61340

[139] Davin McCall and Michael Kölling. 2014. Meaningful Categorisation of Novice Programmer Errors. In *IEEE Frontiers in Education Conference (FIE '14)*. 1–8. https://doi.org/10.1109/FIE.2014.7044420

[140] Davin McCall and Michael Kölling. 2019. A New Look at Novice Programmer Errors. *ACM Transactions on Computing Education* 19, 4 (2019), 1–30. https://doi.org/10.1145/3335814 arXiv:10.1145/3335814

[141] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bull.* 33, 4 (Dec. 2001), 125–180. https://doi.org/10.1145/572139.572181

[142] L. McIver and D. Conway. 1996. Seven Deadly Sins of Introductory Programming Language Design. In *International Conference on Software Engineering: Education and Practice (SEEP '96)*. IEEE Computer Society, Dunedin, New Zealand, 309–316. https://doi.org/10.1109/SEEP.1996.534015

[143] Linda Kathryn McIver. 2001. *Syntactic and Semantic Issues in Introductory Programming Education.* PhD Thesis. Monash University.

[144] R. P. Medeiros, G. L. Ramalho, and T. P. Falcão. 2019. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education* 62, 2 (May 2019), 77–90. https://doi.org/10.1109/TE.2018.2864133

[145] Rolf Molich and Jakob Nielsen. 1990. Improving a Human-computer Dialogue. *Commun. ACM* 33, 3 (March 1990), 338–348. https://doi.org/10.1145/77481.77486

[146] P. G. Moulton and M. E. Muller. 1967. DITRAN - A Compiler Emphasizing Diagnostics. *Commun. ACM* 10, 1 (1967), 45–52. https://doi.org/10.1145/363018.363060

[147] F Mulder. 2016. Awesome Error Messages for Dotty. (Oct 2016). https://www.scala-lang.org/blog/2016/10/14/dotty-errors.html

[148] Christian Murphy, Eunhee Kim, Gail Kaiser, and Adam Cannon. 2008. Backstop: A Tool for Debugging Runtime Errors. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 173–177. https://doi.org/10.1145/1352135.1352193

[149] Emerson Murphy-Hill, Titus Barik, and Andrew P. Black. 2013. Interactive Ambient Visualizations For Soft Advice. *Information Visualization* 12, 2 (2013), 107–132. https://doi.org/10.1177/1473871612469020

[150] Emerson Murphy-Hill and Andrew P. Black. 2012. Programmer-Friendly Refactoring Errors. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1417–1431. https://doi.org/10.1109/TSE.2011.110

[151] Scott Nesbitt. 2017. How to Write Better Error Messages. (Aug 2017). https://opensource.com/article/17/8/write-effective-error-messages

[152] Eric Niebler. 2007. Proto: A Compiler Construction Toolkit for DSELs. In *Proceedings of the 2007 Symposium on Library-Centric Software Design (LCSD '07)*. ACM, New York, NY, USA, 42–51. https://doi.org/10.1145/1512762.1512767

[153] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler Error Messages: What Can Help Novices?. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 168–172. https://doi.org/10.1145/1352135.1352192

[154] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses. In *Proceedings of the 22nd ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, NY, USA, 92–97. https://doi.org/10.1145/3059009.3059026

[155] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, New York, NY, USA, 93–101. https://doi.org/10.1145/2960310.2960316

[156] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 425–436. https://doi.org/10.1145/1993498.1993548

[157] D. N. Perkins and Fay Martin. 1986. Fragile Knowledge and Neglected Strategies in Novice Programmers. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., Norwood, NJ, USA, 213–229. http://dl.acm.org/citation.cfm?id=21842.28896

[158] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 465–470. https://doi.org/10.1145/3017680.3017768

[159] Phitchaya Mangpo Phothilimthana and Sumukh Sridhara. 2017. High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students?. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, NY, USA, 182–187. https://doi.org/10.1145/3059009.3059058

[160] James Prather. 2018. *Beyond Automated Assessment: Building Metacognitive Awareness in Novice Programmers in CS1.* Ph.D. Dissertation. Nova Southeastern University.

[161] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 531–537. https://doi.org/10.1145/3287324.3287374

[162] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/3230977.3230981

[163] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 74–82. https://doi.org/10.1145/3105726.3106169

[164] David Pritchard. 2015. Frequency Distribution of Error Messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. 1–8. https://doi.org/10.1145/nnnnnnn.nnnnnnn arXiv:1509.07238v1

[165] Yizhou Qian and James Lehman. 2017. Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

[166] Keith Quille, Roisin Faherty, Susan Bergin, and Brett A. Becker. 2018. Second Level Computer Science: The Irish K-12 Journey Begins. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18)*. ACM, New York, NY, USA, Article 22, 5 pages. https://doi.org/10.1145/3279720.3279742

[167] Timothy Rafalski, P. Merlin Uesbeck, Cristina Panks-Meloney, Patrick Daleiden, William Allee, Amelia Mcnamara, and Andreas Stefik. 2019. A Randomized Controlled Trial on the Wild Wild West of Scientific Computing with Student Learners. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. ACM, New York, NY, USA, 239–247. https://doi.org/10.1145/3291279.3339421

[168] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. *Electronic Notes in Theoretical Computer Science* 312 (2015), 197–213. https://doi.org/10.1016/j.entcs.2015.04.012

[169] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. 2017. Skalpel: A Constraint-based Type Error Slicer for Standard ML. *Journal of Symbolic Computation* 80 (May 2017), 164–208. https://doi.org/10.1016/j.jsc.2016.07.013

[170] Kyle Reestman and Brian Dorn. 2019. Native Language's Effect on Java Compiler Errors. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. ACM, New York, NY, USA, 249–257. https://doi.org/10.1145/3291279.3339423

[171] Charles Reis and Robert Cartwright. 2004. Taming a Professional IDE for the Classroom. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. ACM, New York, NY, USA, 156–160. https://doi.org/10.1145/971300.971357

[172] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[173] Liam Rigby, Paul Denny, and Andrew Luxton-Reilly. 2020. A Miss is as Good as a Mile: Off-By-One Errors and Arrays in an Introductory Programming Course. In *Proceedings of the 22nd Australasian Computing Education Conference (ACE '20)*. ACM, New York, NY, USA. https://doi.org/10.1145/3373165.3373169

[174] Peter C. Rigby and Suzanne Thompson. 2005. Study of Novice Programmers Using Eclipse and Gild. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '05)*. ACM, New York, NY, USA, 105–109. https://doi.org/10.1145/1117696.1117718

[175] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

[176] Christine Rogerson and Elsje Scott. 2017. The Fear Factor: How It Affects Students Learning to Program in a Tertiary Environment. *Journal of Information Technology Education: Research* 9 (2017), 147–171. https://doi.org/10.28945/1183

[177] Saul Rosen, Robert A. Spurgeon, and Joel K. Donnelly. 1965. PUFFT - The Purdue University Fast FORTRAN Translator. *Commun. ACM* 8, 11 (nov 1965), 661–666. https://doi.org/10.1145/365660.365671

[178] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 598–608. http://dl.acm.org/citation.cfm?id=2818754.2818828

[179] Advait Sarkar. 2015. The Impact of Syntax Colouring on Program Comprehension. *Proceedings of the 26th Annual Conference of the Psychology of Programming Interest Group (PPIG '15)* (2015), 49–58. http://www.ppig.org/library/paper/impact-syntax-colouring-program-comprehension

[180] Thomas Schilling. 2012. Constraint-Free Type Error Slicing. In *Proceedings of the 12th International Conference on Trends in Functional Programming (TFP'11)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

[181] Jean Scholtz and Susan Wiedenbeck. 1993. Using Unfamiliar Programming Languages: The Effects on Expertise. *Interacting with Computers* 5, 1 (1993), 13–30.

[182] Tom Schorsch. 1995. CAP: An Automated Self-assessment Tool to Check Pascal Programs for Syntax, Logic and Style Errors. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '95)*. ACM, New York, NY, USA, 168–172. https://doi.org/10.1145/199688.199769

[183] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, New York, New York, USA, 724–734. https://doi.org/10.1145/2568225.2568255

[184] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag New York, Inc., New York, NY, USA, 672–698. https://doi.org/10.1007/978-3-662-49498-1_26

[185] Dale Shaffer, Wendy Doube, and Juhani Tuovinen. 2003. Applying Cognitive Load Theory to Computer Science Education.. In *Proceedings of the 15th Annual Workshop of the Psychology of Programming Interest Group (PPIG '03)*. 333–346. http://ppig.org/library/paper/applying-cognitive-load-theory-computer-science-education

[186] W. J. Shaw. 1989. Making APL Error Messages Kinder and Gentler. In *Conference Proceedings on APL As a Tool of Thought (APL '89)*. ACM, New York, NY, USA, 320–324. https://doi.org/10.1145/75144.75188

[187] Ben Shneiderman. 1982. Designing Computer System Messages. *Commun. ACM* 25, 9 (1982), 610–611. https://doi.org/10.1145/358628.358639

[188] Ben Shneiderman and Catherine Plaisant. 2004. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (4th ed.). Pearson Addison Wesley.

[189] M. E. Sime, A. T. Arblaster, and T. R. Green. 1977. Structuring the Programmer's Task. *Journal of Occupational Psychology* 50, 3 (sep 1977), 205–216. https://doi.org/10.1111/j.2044-8325.1977.tb00376.x

[190] Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. 2015. Analyzing Student Work Patterns Using Programming Exercise Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 18–23. https://doi.org/10.1145/2676723.2677297

[191] Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 283–299. https://doi.org/10.1145/2661136.2661156

[192] Andreas Stefik, Bonita Sharif, Brad. A. Myers, and Stefan Hanenberg. 2018. Evidence About Programmers for Programming Language Design (Dagstuhl Seminar 18061). *Dagstuhl Reports* 8, 2 (2018), 1–25. https://doi.org/10.4230/DagRep.8.2.1

[193] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education* 13, 4, Article 19 (Nov. 2013), 40 pages. https://doi.org/10.1145/2534973

[194] S Suhailan, S Abdul Samad, and M A Burhanuddin. 2014. A Perspective of Automated Programming Error Feedback Approaches in Problem Solving Exercises. *Journal of Theoretical and Applied Information Technology* 70, 1 (2014), 121–129. http://www.jatit.org/volumes/Vol70No1/16Vol70No1.pdf

[195] John Sweller. 1988. Cognitive Load During Problem Solving: Effects on Learning. *Cognitive science* 12, 2 (1988), 257–285. https://www.sciencedirect.com/science/article/pii/0364021388900237

[196] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, and Matthew C. Jadud. 2011. Predicting At-risk Novice Java Programmers Through the Analysis of Online Protocols. In *Proceedings of the 7th International Workshop on Computing Education Research (ICER '11)*. ACM, New York, NY, USA, 85–92. https://doi.org/10.1145/2016911.2016930

[197] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (sep 1981), 563–573. https://doi.org/10.1145/358746.358755

[198] Warren Teitelman and Larry Masinter. 1981. The Interlisp Programming Environment. *Computer* 14, 4 (1981), 25–33. https://doi.org/10.1109/C-M.1981.220410

[199] Emillie Thiselton and Christoph Treude. 2019. Enhancing Python Compiler Error Messages via Stack Overflow. In *Proceedings of the 19th International Symposium on Empirical Software Engineering and Measurement (ESEM '19)*. arXiv:1906.11456. http://arxiv.org/abs/1906.11456

[200] Suzanne Marie Thompson. 2006. *An Exploratory Study of Novice Programming Experiences and Errors*. Masters Thesis. University of Victoria.

[201] Warren Toomey. 2011. Quantifying The Incidence of Novice Programmers' Errors. (2011), 5 pages. https://minnie.tuhs.org/Programs/BlueJErrors/arjen_draft.pdf

[202] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010, Article 3 (Jan. 2010), 26 pages. https://doi.org/10.1155/2010/602570

[203] Kota Uchida and Katsuhiko Gondow. 2016. C-Helper: C Latent-error Static/Heuristic Checker for Novice Programmers. In *Proceedings of the 8th International Conference on Computer Supported Education (CSEDU 2016)*. SciTePress - Science and Technology Publications, Lda, Portugal, 321–329. https://doi.org/10.5220/0005797703210329

[204] Miguel Ulloa. 1983. A survey of run-time and logic errors in a classroom environment. *ACM SIGCUE Outlook* 17, 3 (1983), 21–25. https://doi.org/10.1145/1045078.1045081

[205] Leo C. Ureel II and Charles Wallace. 2019. Automated Critique of Early Programming Antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 738–744. https://doi.org/10.1145/3287324.3287463

[206] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. 2014. No Tests Required: Comparing Traditional and Dynamic Predictors of Programming Success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 469–474. https://doi.org/10.1145/2538862.2538930

[207] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2012. BlueFix: Using Crowd-sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In *Proceedings of the 11th International Conference on Advances in Web-Based Learning (ICWL'12)*. Springer-Verlag, Berlin, Heidelberg, 228–239. https://doi.org/10.1007/978-3-642-33642-3_25

[208] Richard L. Wexelblat. 1976. Maxims for Malfeasant Designers, or How to Design Languages to Make Programming As Difficult As Possible. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 331–336. http://dl.acm.org/citation.cfm?id=800253.807695

[209] Emily Wilska. 2004. Non-Fatal Errors : Creating Usable , Effective Error Messages. http://www.writersua.com/articles/message/index.html. (2004).

[210] Niklaus Wirth. 1968. PL360, a Programming Language for the 360 Computers. *J. ACM* 15, 1 (Jan. 1968), 37–74. https://doi.org/10.1145/321439.321442

[211] Alexander William Wong, Amir Salimi, Shaiful Chowdhury, and Abram Hindle. 2019. Syntax and Stack Overflow: A Methodology for Extracting a Corpus of Syntax Errors and Fixes. (Jul 2019). arXiv:1907.07803 http://arxiv.org/abs/1907.07803

[212] John Wrenn and Shriram Krishnamurthi. 2017. Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 134–147. https://doi.org/10.1145/3133850.3133862

[213] Baijun Wu, John Peter Campora III, and Sheng Chen. 2017. Learning User Friendly Type-error Messages. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 106 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133930

[214] Stelios Xinogalos, Maya Satratzemi, and Vassilios Dagdilelis. 2006. An Introduction to Object-Oriented Programming with a Didactic Microworld: objectKarel. *Computers and Education* 47, 2 (2006), 148–171. https://doi.org/10.1016/j.compedu.2004.09.005

[215] Stelios Xinogalos, Maya Satratzemi, and Christos Malliarakis. 2017. Microworlds, Games, Animations, Mobile Apps, Puzzle Editors and More: What is Important for an Introductory Programming Environment? *Education and Information Technologies* 22, 1 (Jan. 2017), 145–176. https://doi.org/10.1007/s10639-015-9433-1

[216] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '99)*. Springer-Verlag, Berlin, Heidelberg, 253–267. http://dl.acm.org/citation.cfm?id=318773.318946

[217] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/2737924.2738009

[218] Lu Zhang. 2012. *Empirical Design and Analysis of a Defect Taxonomy for Novice Programmers*. Masters Thesis (by Research). The University of Western Australia. https://research-repository.uwa.edu.au/files/3232803/Zhang_Lu_2012.pdf

[219] Daniel Zingaro, Michelle Craig, Leo Porter, Brett A. Becker, Yingjun Cao, Phill Conrad, Diana Cukierman, Arto Hellas, Dastyni Loksa, and Neena Thota. 2018. Achievement Goals in CS1: Replication and Extension. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 687–692. https://doi.org/10.1145/3159450.3159452