Title: An adaptive feedback framework for a language-independent intelligent programming tutor (IPT) using ANTLR

Opeoluwa Joseph Ladeinde (M2108909)

Supervisor: Dr Mohammad Abdur Razzaque

School of Computing, Engineering and Digital Technologies
Teesside University, UK

The thesis is submitted for the degree of
Doctor of Philosophy

July 2023

I would like to dedicate this thesis to my, supervisor, my family, and the online communities I'm a part of aspiring to program, or with an interest in Game Design.. . .

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This thesis contains fewer than 40,000 words including appendices, bibliography, footnotes, tables and equations.

Opeoluwa Joseph Ladeinde (M2108909)
July 2023

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr Mohammad Abdur Razzaque for their invaluable advice, continuous support, and patience during my PhD study. I am also grateful to Dr Razzaque's for his guidance and support during this journey. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. Additionally, I would like to give special thanks to Junior Ladeinde for his tremendous knowledge on literacy, grammar and writing. I would also like to thank Dr Julien Cordry and Dr Riazul Islam for the revision recommendations. Finally, I would like to express my gratitude to my parents and my sister. Without their tremendous understanding and support, it would be impossible for me to complete my study.
2 Timothy 16-17

Abstract

Intelligent Tutoring Systems specifically for programming and computer science can be referred to as Intelligent Programming Tutors. Currently, the only consistent component in Intelligent Programming Tutors is the fact they provide Adaptive Feedback. However, the methods IPT provide are created specific to both their educational context, and programming language - they are domain specific. Several components of the domain of IPT are considered mutually exclusive - particularly the selection of programming languages each IPT can support, and the authoring tools that can be made for them.

This research introduces a method of program synthesis with a domain-independent IPT (programming-language-independent IPT) method of specifying programming language syntax, and outputting syntax feedback and task semantics feedback according to the given programming language. This method also introduces an authoring tool that can be used across all theoretical text based programming languages.

We use ANTLR4 for the program synthesis behind this IPT. We test our method against a small sample of the ever growing repository of more than 200 programming languages. We conclude that the method worked unanimously across programming languages - allowing for an IPT that guarantees feedback is generated, regardless of programming language, or the state of the learner's code and any errors in the solution. However, what affects the quality of the feedback returned, and what would be ideal for phrasing is still ambiguous.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

This section consists of the introduction of this thesis, consisting of an overview of Intelligent Programming Tutors, feedback, self-explanation, and ANTLR4. The section is ultimately used to:

- state the particular teaching subject this research is applied to (computer science and programming),
- define to the reader various forms of pedagogy that will be discussed through this thesis, and
- list critical features of a compiler generator named "ANTLR4" for use in referring to programming languages by abstraction.
- list the aims and contributions of this research on Intelligent Programming Tutors

1.1 Defining: Intelligent Programming Tutors

Tutoring in programming related courses (and Computer Science) is notably challenging for tutors to promote learning in education. For instance, Caspersen notes various cognitive science and educational psychology patterns that should be observed for each learner when designing programming courses[3]. Higher education pass rate can consistently be quantified to be low, with Watson's 2014 study, stating a 66.4% pass rate for introductory programming courses[4]; reaffirming the findings of a study 7 years ago by Bennedsen et al. [5]. Learners can easily fall within the lower bounds of these rates, without receiving effective tutoring, effective learning environments or deploying self tutoring tactics.

There are many potential contributing factors to these rates. It partially can be attributed to challenges in allocating time to learners when they need assistance (or discerning when

they need assistance[6]). Additionally, there is an encouraged need to help learners identify relatable uses for what feedback is given to learners, and what they learn, [7, 8].

For these two problems, a number of automated tools, software, online environments and learning environments have been created to support human tutors.

"Intelligent tutoring systems" (ITS) commonly refers to automated systems designed to provide adaptive, or personalised learning for any given subject. The semantics on what within the systems should be adaptive (ultimately, the intelligence of the ITS) can differ between primary and secondary studies. Regardless of how the "intelligence" and "adaptability" of ITS is defined, ITS and studies on ITS share at least one common aim. ITS often aim to present learning content and feedback that is relative to the skill, aims, knowledge or problems of the learner using the system. Ultimately ITS commonly aim to personalise the tutored content to the learners. Such aims are commonly observed in secondary studies on programming based ITS [1, 9–12].

Such systems need to form contextual feedback responses to learner's actions (providing feedback for tasks or guiding menu navigation).

The automated personalising of learning has been performed in the field of intelligent tutoring systems (ITS). Several ITS already exist, with a number arguably being shown to help improve test rates in *strictly quantitative*[1] studies such as Kulik et al.'s meta analysis [13] (compared to no ITS aided tutoring).

For automated *tutoring* of programming, we have ITS for programming; "Intelligent Programming Tutors" (IPT) (a term used in a review by Crow's[10]). IPT refers to tutoring systems that facilitate learners attempting to answer programming activities and automatically responding with feedback (or hints) specific to what the learners aim to perform (or improve). For *assessing* the programming of learners, we have "Automated Assessments Tools" (AAT) for programming. This term appears in reviews such as Douche et al. [14] and Pettit's critique on AAT research[15].

The former (IPT) is the focus of this paper. Due to this, it is worth distinguishing it from AAT, which are more likely to provide "summative" feedback [2], over the "formative" feedback[3] ideally contained in IPT. This research will still refer to AAT when relevant to discuss when researched alongside IPT. The difference between the two are illustrated in Figure 1.1.

[1] I critically discuss the viability of quantitative studies for observing the benefits of particular pedagogy in Chapter 2.

[2] feedback relative to obtaining a grade

[3] feedback relative to the learner improving or reaching their aims

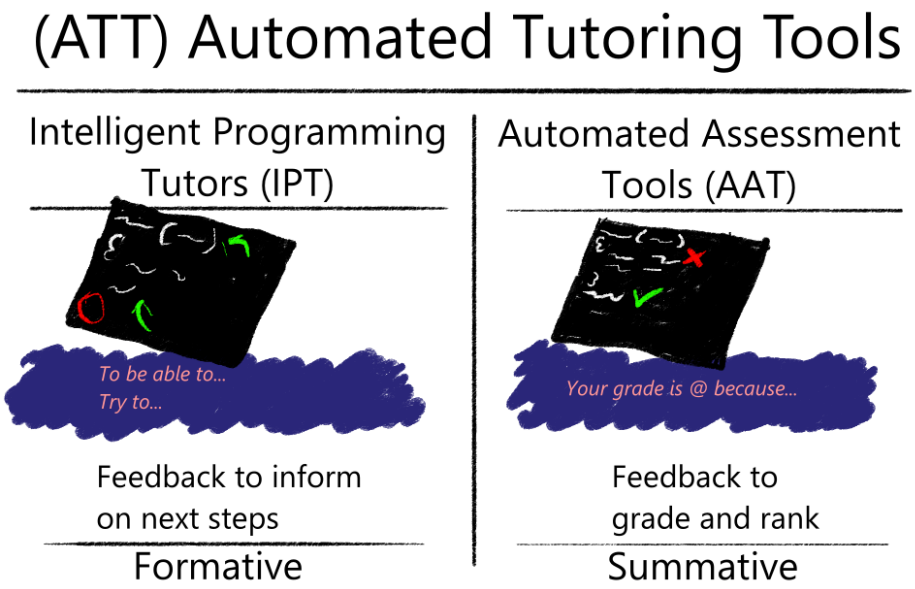

Fig. 1.1 An illustration of the differences in feedback between the two Automated Tutoring tool (ATT) types: Intelligent Programming Tutors (IPT) and Automated Assessment Tools (AAT)

1.2 Pedagogy in Intelligent Programming Tutors

Primary studies on IPT have noted achievements of feats, such as:

- Returning relevant feedback to learners in programming tasks, regardless of how broken the learner's code is [16, 17].
- Programming task authoring tools that enable responses to specific problems by writing an inline comment [18].
- Generate and personalise tasks to suit what the system knows of the learner [19].
- Suggestions on which IPT content the learner should approach, based on the content the learner has attempted, and a database on what content requires skills from other content [20].
- A multi-user programming environment that facilitates peer-learning [21] (defined by educational psychology papers such as [22]).
- A simulated student within a multi-user programming environment, played by the IPT [23].

IPT has also successfully facilitated self-explanation (and observed its benefits) in environments extraneous to their programming environment[24, 19].

However, it is critical to note that within the field of IPT, there are inconsistencies problems in how learning is observed (and hence how IPT can be modelled around learning), how the creation and investigation of IPT are programming language specific and domain-specific, and an overall prevalence of non-standardised observations on the field. This research confirmed these observations on the field through Chapter 22.1.

Each of the IPT achievements listed above could be argued to be completely independent of each other - rarely contained within the same IPT. The exceptions are IPT primary studies directly improving on themselves. This is observed in secondary studies containing IPT such as Keuning's [25, 10, 1].

Two secondary studies that note mutually inclusive paradigms that were not combined were:

1. Keuining et al. [25] named 18 "feedback types", 8 general "techniques" for providing feedback, 5 classifications of "adaptability" for task creation, and 5 noted "quality" of primary study research, each with AA or IPT that had aspects that did not fall into the named categories. Keuining's study also recorded which concepts were most often overlapped.

2. Crow [10] concludes, "it is evident that there is no standard combination of features that have been utilised within the field of intelligent tutoring applied to programming education". They also evaluate "studies could focus on finer grained measurement of the usage and effectiveness of individual features within IPTs or comparing different implementations of systems".

My research further discusses a lack of integration in research in Section 2.1, where a literature review is performed, noting 11 of 14 secondary studies at least noting a lack of integration in the field.

This implies a potential lack of consensus - even on the core pedagogy that inspires the methods used within the IPT and how they are assessed empirically. This can be observed in contrasting opinions on how the improved 'quality of learner' is assessed by papers observing immediate test results (such as Nesbit [26] and Mousavinasab [11]) and papers finding contextual differences to learning, as well as discussing the lack of pedagogical discussion (such as Zawacki's [27] and Harley's [28]). Mousavinasab's paper, in particular, discusses varying evaluation methods across the field [11].

Success within the field can superficially be observed through test scores, as accomplished by research such as Kulik's [13]. However such work (and many of the primary works

documented within by the restrictions of the inclusion criteria) are purely quantitative. Observations on any learning that has occurred, could be argued to be incidental to the tests, rather than evaluating the learner's long-term application of the tutored concepts (Kulik's paper for instance, could only note 2 publications with to be more than 2 sessions). For instance, after being observed in application over 10 years, the widely used ALEKS [29] (spanning 15 empirical studies between 2005 and 2015) was argued to be not be objectively better than traditional classroom teaching. Also, the majority of IPTs are limited to tutoring introductory programming, and many evaluations in IPT primary works may be limited to a single-day course and test.

Success within the field can also be measured by evaluating the proposed cognitive benefits in educational psychology and theoretical design papers, such as for instance, Caspersen's [3] design theory. For learning to be personalised to the learner, it's critical for cognitive and educational psychology concepts to be observed [8, 3], which in turn could enable their proposed benefits to be observed. Unfortunately, there's no standardised pedagogy or discussion for this across IPT primary studies. However, modelling ITS on constructivism [11, 30], and around cognition [31, 28] is a known practice, as is evaluating the proposed benefits of the respective theories.

Overall, even when evaluating how one can define how to observe the "quality of learning" changes between studies. Our research bases our observation on the "quality of learning" on constructivism. This thesis will observe the quality of learning as: how well learners retain the memory of concepts and how easily learners can identify solutions to concepts or problems.

It's a challenge to have automated tutors provide feedback with the same potential as human tutors. Nguyen defines 5 levels of activity 'definedness' [32], where only relatively recently (2017) has an automated tutor for programming been able to work with class 4 definition of activities (activities that support multiple solution strategies that can not all be preset by the automated tutor). Hence, there are presently tasks that are presently limited to a domain, and tasks that can not be defined and observed by automated tools, to return feedback for.

We discuss feedback definedness in Section 2.2, and an IPT that supports class 4 definition of activities (the Intelligent Teaching Assistant for Programming or ITAP by Rivers [16]) in Section 3.

In the following subsections, I discuss to what extent feedback and self-explanation have been explored in IPT and present problems in how these pedagogical concepts are applied in IPT.

1.2.1 Pedagogy - Feedback

One pedagogical area of critical importance is feedback in program-related courses. The timing of *when* feedback is returned to the learners is important to consider, along with *how much* information on the solution can be given [30]. The impact feedback has on learning is often observed in empirical studies on automated tutors (which include IPT). This can be observed in primary studies such as River's IPT [17], which was adapted to contain more information in later iterations. Adding information to feedback improved the motivation of the learners. There also exists at least two secondary studies on the feedback in automated tutors, Keuning et al's [25] and Nguyen's [1] (discussed further in section 2.2). Koedinger also raises the ambiguity of when hint-based feedback should be used in ITS [30]. If feedback is inadequate[4], it can have adverse effects on learning.

Hence, it would not be contentious to state that feedback is more effective if it is:

- responsive to the request,
- informative on what the problem is,
- contains the right amount of information to guide resolving the problem,
- adapted to the learner, referring to what they tried and how to reach their aim relative to that,

Each IPT and AAT differ in the context they are made to support, and what their respective primary studies aim to achieve (often to improve learning in their definition[5] of learning). They ultimately differ in how they're implemented and what they innovate over. Keuning et al. note more than 110 AAT and IPT in their study [25].

This extends to how adaptive feedback[6] differs across IPT.

Empirical studies surrounding IPT and AAT vary in:

- how they view learning, the aims of what the tool intends to do to 'improve learning',
- the contexts within computer science or programming they tutor in,
- and ultimately how they return feedback.

[4] poorly timed or lacking in constructive information to help the learner learn how to reach their aims

[5] Learning pedagogy can often differ between learning tools. However, each primary study's pedagogy is often stated. Differing pedagogy between primary studies is often noted in secondary studies of IPT.

[6] "Adaptive Feedback" is the feedback that is dynamically phrased to refer to what a learner attempted, and how they can reach the activity aims. The term appears in secondary studies on IPT, such as Nguyen's [1] and Crow's [10]

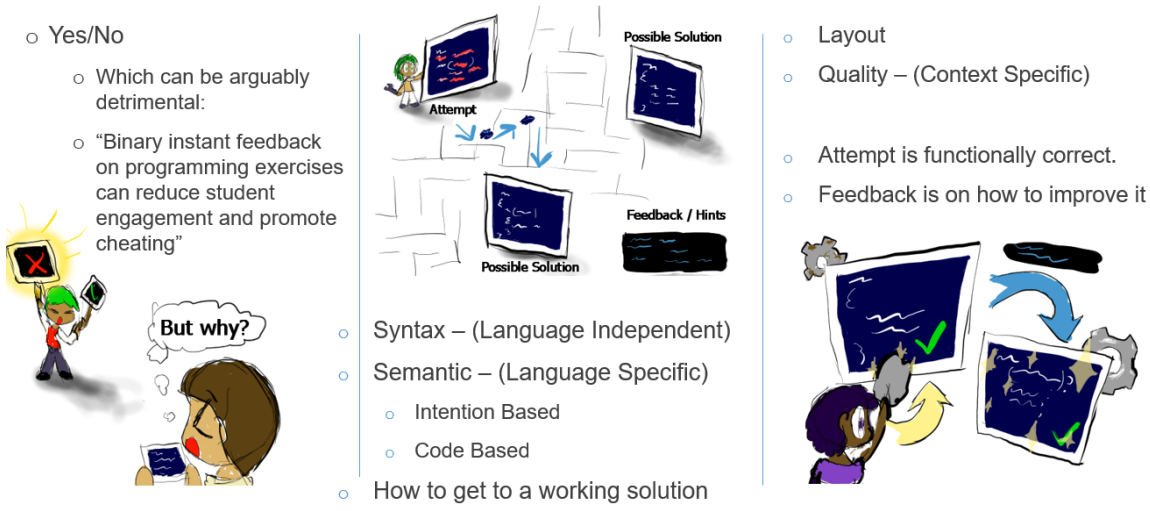

Fig. 1.2 Illustrations of the 5 categorised forms feedback may take according to Nguyen's [1]. A quotation is used from Kyrilov [2]

Hence, along with different methods of returning adaptive feedback, automated tutoring tools overall differ in what feedback they return, and when they can support returning feedback. Secondary studies such as Keuning's [25] provide a detailed evaluation and classification of AAT and IPT, further discussed in Section 2.2. A recurring discussion through IPT primary studies [24, 16, 33–39] is hint generation, as a form of feedback.

Along with different methods and pedagogy, IPTs vary in what context they work for, often being limited to one programming language. This results in each IPT being domain-specific, even if a number of their theories could be reconstructed for different IPTs from the ground up - even among hint generation for feedback.

Hence, the central aim of our research was on establishing what components of an IPT could be domain-independent. I explore: in IPT could a state-of-the-art method for forming feedback, be recreated to be domain-independent? In the following section, we discuss another method of pedagogy - self-explanation. We also discuss how pedagogy is observed in the field of IPT.

1.2.2 Pedagogy - self explanation

Self-explanation can be defined as the process of a learner redefining[7] of a concept. This in turn has the learner cognitively work a concept, which ultimately helps the learner memorise concepts, and helps learners apply concepts.

[7] reciting, listing, restructuring through notes, rewriting or phrasing their definition - definitions related to generative learning defined by Fiorella [40]

In computer science (as well as other educational contexts), studies have attempted to observe the effect of promoting self-explanation between tasks and information. Paron et al.'s study [41] contained an immersive virtual reality learning environment for tutoring hard to grasp biology concepts. They proposed the facilitation of self explanation would improve the performance in a post-activity test.This was observed to produce noticeably better post-test results compared to the group without facilitated self-explanation. The test was taken immediately after the lesson. However, the encouraged self-explanation (through note-taking) took place outside of the immersive virtual reality learning environment. The test was also limited to a very specific, single-lesson context. This is also noted by the authors, "A limitation of this study is that it involved one, short off-the-shelf instructional lesson delivered in a lab setting with an immediate test". Ultimately, the study only covered the short-term effects of lessons and results.

Encouraging self-explanation has been discussed in e-learning for computer science by Garcia [42], but has also been attempted in at least one IPT; notably in Marwan's primary study [24] on the IPT iSnap. In this work, requesting users to define what they tried was presented in a separate string box underneath the hint asking "Why do you think iSnap recommended this hint?" Unlike the previously mentioned work, the encouraging self-explanation was internal to the IPT itself, but like the previously mentioned work, it was still separate from the programming interface. The study measured the time taken to complete tasks. In order of compared completion times for hint tests:

1. Time with no hints had the longest completion times.
2. Less code hints had shorter completion times.
3. Code hints with self-explanation prompts had the shortest completion times.

They additionally evaluated: "Future work should investigate the hypothesis that code hints with both textual explanations and self-explanation prompts may be more effective for helping students to repeat things they have already done than completing new tasks". This conclusion was drawn due to the result of self-explanation prompts having less of an impact on new or unrelated tasks.

These two examples work to reaffirm the theorised benefits that self-explanation improves memory retention of concepts and applying such concepts. They also imply two unknowns. These are on the long-term effects of self-explanation on tasks, and one proposed by my research: can self-explanation be observed, integrated and encouraged in an IPT's activity framework (rather than a through a mutually exclusive interface)?

Within programming, "code comments" (the act of writing notes in code) can require users to structure their intention in the programming interface itself. In this regard, my

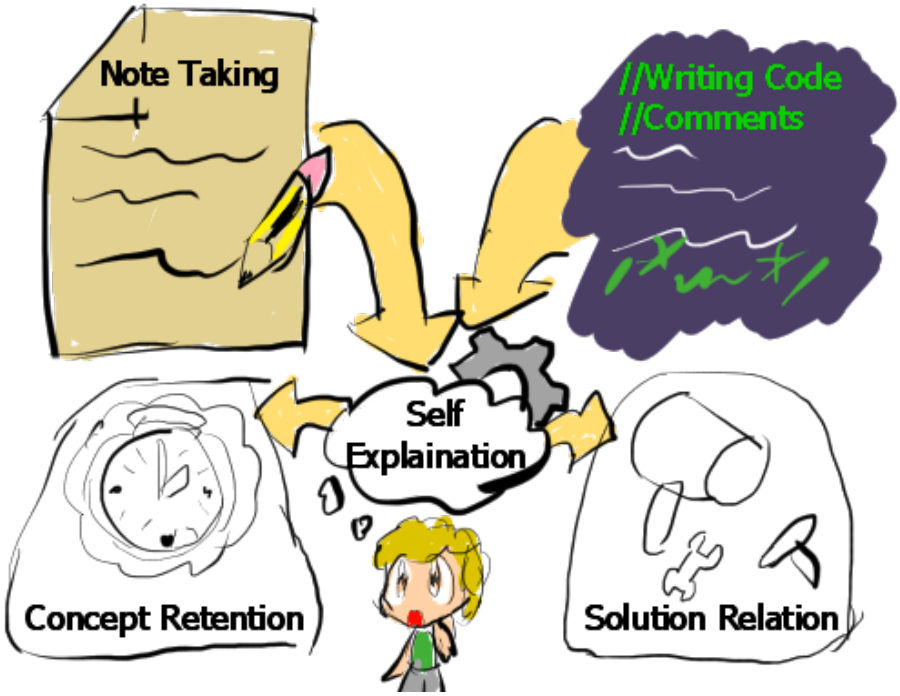

Fig. 1.3 An illustration on the cognitive effects of self-explanation, and the proposed effects of code comments on learning. Self-explanation elicits memory retention of concepts noted and deep learning of those concepts.

research hypothesises it is possible to elicit self-explanation through code comments. By being within the programming interface (a non extraneous component to programming) self-explanation can be encouraged habitually. My research also hypothesises that by being integrated into the code, an IPT can observe where self-explanation has or has not occurred, where it should occur, and hence be able to provide feedback to encourage self-explanation when appropriate.

Hence, our research has two questions concerning facilitating self-explanation in IPT:

- "Can self explanation be facilitated through code comments?" We hypothesise that the act of promoting self-explanation through code comments would elicit the same benefits of code comments (illustrated in Figure 1.3. I also hypothesise that it would also promote a self-regulation strategy to self-explain through code comments (a persistent feature outside of IPT and learning contexts), and could be used to personalise the prompts for code comments.

- "When does promoting self explanation have an impact on the active task and subsequent tasks?". This is in direct response to tests of self-explanation being short-term, and an evaluation question proposed by Marwan's study [24].

These questions served as key aims to the direction taken by my research. A number of investigations were taken to allow answering these questions thoroughly across IPT, for all possible programming languages. These investigations (covered in the next Section, Section 1.3) in themselves became the main contributions of this thesis. My research had additional aims to be used to answer the questions on: 'when code comments can elicit the benefits of self explanation, and enable IPT to observe self explanation'.

One question was finding to what degree pedagogy surrounding code comments could be applicable across programming languages. A large number of programming languages support a method that enables the person writing code to write notes/text (to themselves or other users) that do not affect compilation.

Hence, we proposed investigating how often facilitated code comments are in ANTLR4 programming language grammars, through the question: (Research Question 2d)"What is the frequency of grammars including, streaming or excluding (skipping) code comments?". We have found to present date, one ([43]) work which also explored this question which we will discuss in Section 2.3.1.

Answering this question was needed to answer for all supporting languages: "Does encouraging the use of code comments elicit the same benefits as self explanation?" which will be a question for a subsequent study.

The follow-up questions on ANTLR4 are first defined in section 1.3, before being investigated through this thesis.

My research intended to answer this question theoretically for every possible programming language. However, to the date of this thesis, no other IPT outside our research project can refer to programming languages by abstraction. Hence my research pursued the unique contributions of proofing a state-of-the-art IPT method for all programming languages, and potential phrasing restrictions that could prevent the application of code comments to promote self-explanation. Ultimately, while "can code comments elicit the same benefits as self explanation" is not answered in this thesis, it served as the motivation for introducing the knowledge the answer this question: language-independent IPT. It also motivated several investigations towards answering that question, such as: "What is the frequency of grammars including, streaming or excluding (skipping) code comments?" covered in the following section.

1.3 A solution for language integrated techniques

The aforementioned secondary studies highlight the exceptional contributions to the field of IPT. However, they also note that IPT research has differing focuses that often do not include

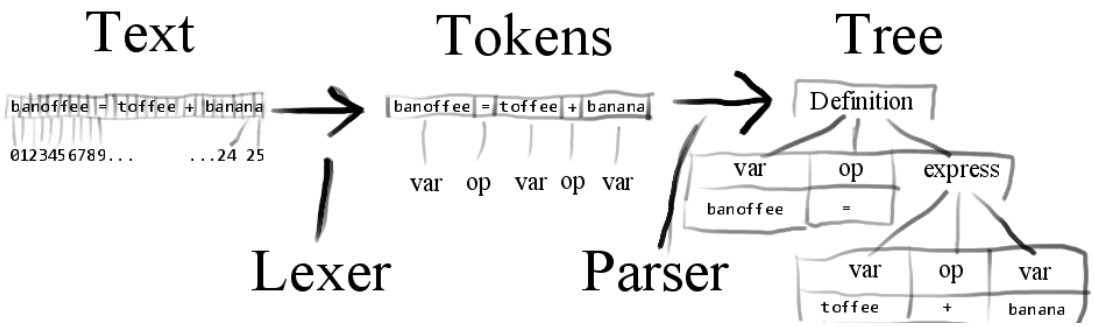

Fig. 1.4 A diagram displaying the change in representation from raw code text, to tokens (via a tokeniser), to an abstract syntax tree (via a parser).

the contributions of each other. For instance, ITAP (an IPT composed by [17] for Python) is uniquely capable of returning adaptive feedback regardless of the state of the learner's code, and adding solution strategies using working solutions. This IPT is also capable of returning adaptive feedback that acknowledges there can be multiple solution strategies that can lead to the answer. This solution is currently limited to Python. The method used for task creation is also specific to ITAP. With the majority of IPTs being surveyed to focus on one language or pedagogy, it is common for IPT implementation methods to be domain-specific.

Because of the non-integrated nature of IPT research (differing languages, implementation methods and learning activity creation methods), innovation through IPT is often limited to their specific context. This, in part, can be attributed to the fact that programming languages differ in grammar and semantics. However, text-based programming languages themselves have an explicit definition that forms rules across all text-based programming languages. In theory, these rules can be referred to by abstraction. This has yet to be attempted within the field of IPT. Tokenizer[8] and parser[9] generators provide ways to define language rules, refer to them by abstraction, and create compilers for the language. The process of tokenising and parsing on code is illustrated in Figure 1.4.

Furthermore, there is little support for an IPT framework to create and use custom programming languages easily. Custom programming languages can be used to aid in teaching core concepts that are needed to utilise more common languages fully. For instance, works such as Cervesato et al.'s [44] designed and used a custom language called "C0" to introduce learners to C++ in introductory programming.

[8]Also referred to as lexing, it is the process of converting a series of text characters into legible words for a given programming language's rules. These formed series of characters are called "Tokens"

[9]The process of forming structures or phrases from a series of tokens according to the rules of a given programming language. This process forms an abstract syntax tree (AST) - a full representation of what refers to what within code

Parr's ANTLR4 is such[10] a tool [45] with seemingly no limits on the programming languages that can be defined through its state-of-the-art variant of LL (given the language is purely or functionally text-based) [46]. ANTLR4 also has a well-revered public domain for people to create grammar - from well-known programming languages to custom languages for a specific tutoring context. Figure 1.5 contains a snapshot of this public domain. ANTLR4 can theoretically also return syntax errors for all code submitted under a language.

Using ANTLR4, my research investigated what became my first key question: "to what degree a programming language-independent IPT's feedback can provide the same quality of learning as state of the art IPT". A hypothesis of mine was that programming language-independent IPT could be formed (using state-of-the-art feedback methods) by referring to syntax by abstraction. I investigated whether an IPT (adaptive feedback framework, task creation and adaptive navigation) can be constructed through abstraction:

- Research Aim 1a Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?
- Research Aim 1b Are there instances of language-specific additions needed to return feedback to a language-independent IPT?

It is worth noting that ANTLR4, compared to state-of-the-art language-specific compilers (such as Medeiros' for Lua [47]) ANTLR4 has compile times that can be up to 6 times slower on average. When compared to Medeiros, it had a few instances of additional errors highlighted relative to Medeiros' compiler. Although for Medeiros' test, we can not say for absolute certainty if any drawbacks on ANTLR4 were due to how the Lua grammar was defined. However, the critical hypothesised difference in error message feedback and compile time was due to ANTLR4's algorithm, so grammar variants could still yield similar differences or worse results. Regardless, although ANTLR4 has a chance of being less efficient than state-of-the-art language-specific compilers, it is still reliably functional for its purpose in generating compilers for theoretically any imagined language [46] and successfully parsing error code.

This research sought to determine what would be the second key question, "what impact can grammar phrasing have on the quality of learning". Another hypothesis of mine was grammar semantics could affect the legibility of feedback and how easily code comments could be read by the compiler. Hence it was critical to investigate the types of differences ANTLR4 grammar variants can have on potential feedback (phrasing and speed of feedback within the IPT), and potential use to observe and prompt self-explanation within IPT:

[10] compiler generator that can be referred to by abstraction

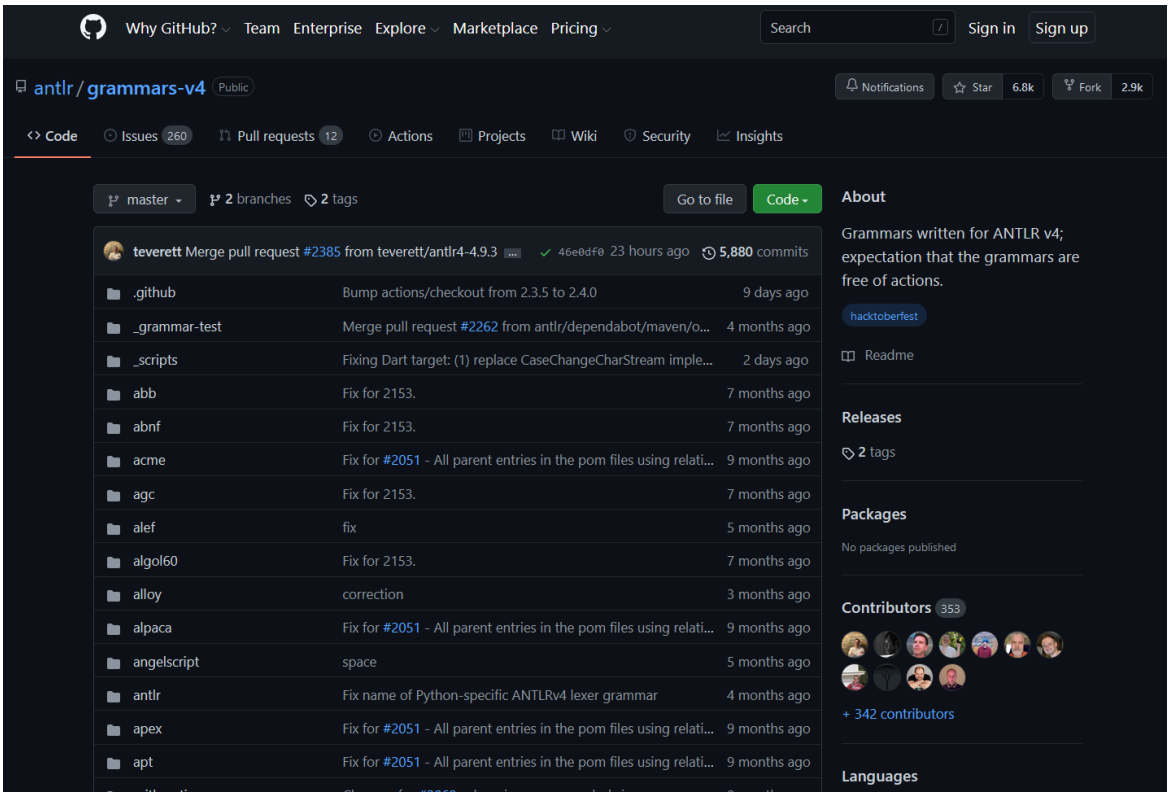

Fig. 1.5 An image of the GitHub repository for ANTLR grammars

- Research Aim 2a: What are the root types that need to be called for each lexer and parser? What types need to be called in sequence?
- Research Aim 2b: What problems in grammar definitions can be listed/categorised to make feedback implicit? How can the phrasing of grammar affect the quality of feedback?
- Research Aim 3a: What conditions in grammar definitions can be listed/categorised to affect the speed feedback can be returned? How does the compile speed change for different contexts?
- Research Aim 3b: What is the frequency of grammar including, streaming or excluding (skipping) code comments?

This research was also inspired by a lack of state-of-the-art methods being implemented in IPT for C++, Lua and C# (of IPT existing for such languages at all). We propose the use of ANTLR4 to implement a solution of a similar premise to ITAP for all languages in IPT, which could investigate the versatility of state-of-the-art solutions in IPT for every

programming language (existing). We will also sample how functional the grammars on the ANTLR4 repository are with the solution we propose in an IPT.

1.4 Research Aim and Objectives

To reiterate, to explore what pedagogy can be implemented across IPT, to explore what quality of learning can be achieved in a language-independent IPT, and to explore any potential restrictions to evaluating an IPT that facilitates self-explanation with code comments, my research investigated the following core questions:

	Research Aim (RA)	Research Objective (RO)
1	'To what degree a programming language-independent IPT's feedback can provide the same quality of learning as state-of-the-art IPTs?	To answer question 1 in general, I constructed a language-independent IPT and performed a technical analysis to see if it can produce feedback/hints with similar parameters to River's [17] IPT, for all possible learner's code (which was first unique to ITAP), for all programming languages (which is unique to our research). This IPT was then sent with a simple activity to "follow the instructions to complete the code" for randomly selected programming languages (out of a pool of 15) to confirm the integrity of the rubric this research used for the technical analysis.
1a	Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?	In particular, RA1a confirms if the method described in this thesis proves a method to create River's [17] unique contributions for any programming language, but *also* within the same IPT.
1b	Are there instances of language-specific additions needed to return feedback to a language-independent IPT?	This question facilitates discussion on potential language-specific constraints to find how language-independent a language-independent IPT can be; the potential limitations of this method. To answer this, we list the number of languages requiring more than one root token and those requiring normalising variables. This research also discusses when languages specific constraints apply to syntax, semantic, quality and layout feedback (4 types of IPT feedback listed by Nguyen [1].

	Research Aim (RA)	Research Objective (RO)
2	'What impact can grammar phrasing have on the quality of learning?'	To answer question 2 in general, a source containing multiple ANTLR4 language grammars is surveyed.
2a	'What are the root types that needs to be called for each lexer and parser? What types need to be called in sequence?'	RA2a is inquired as programming languages can sometimes require multiple passes or contain a different root type [46]. This question is used to investigate if there are potential exceptions to when a programming language can be used (given that the grammar4 of a given language compiles successfully). We sample ANTLR4 grammars and list root token types, used token types, an exception to when one token type suffices and a method to resolve that exception.
2b	'What problems in grammar definitions can be listed/categorised to make feedback implicit? How can the phrasing of grammars affect the quality of feedback?'	The critical question is RA2b, which inquires how the phrasing of a grammar file can affect feedback. The use of the token names within an application using ANTLR4 has not appeared in empirical research on ANTLR4. We sample the phrasing of the grammar documents themselves. We construct a rubric, assign each grammar a "legibility" description, and then give users a test build of ANTLR4 to see if their questionnaire responses correspond to the legibility assigned by our rubric. This provides knowledge on what affects the legibility of feedback produced from ANTLR4 grammars and how ANTLR4 grammars on the repository are legible. Additionally, the questionnaire is also used to survey what part of the feedback is more helpful to the learners (for the given language's phrasing).

	Research Aim (RA)	Research Objective (RO)
3	'What impact can grammar phrasing have on the quality of learning?'	To answer question 3 in general, a source containing multiple ANTLR4 language grammars is surveyed.
3a	'What conditions in grammar definitions can be listed/categorised to affect the speed feedback can be returned? How does the compile speed change for different contexts?	As it was unclear to what degree ANTLR4's grammar's phrasing can affect compilation speed (and to provide an estimate to language-specific compiler's speeds), RA3a is used to give more results on the subject of speed of returned feedback. The idea was to compare the phrasing of the different grammar of the same language. As there was a lack of variants of languages on the repository, this research provides feedback speed by code size and language. This is obtained from the user-distributed copy of **Nue** Tutor (proposed in this work). However, we document the method for the research to be enacted more thoroughly,
3b	'What is the frequency of grammars including, streaming or excluding (skipping) code comments?'	Finally, RA3b provides knowledge on when a language-independent IPT is able to read code comments. This is simply found by surveying the grammar for how they handle the respective comment token type. This is confirmed as code comments can often be discarded during compilation. This confirms how common it is for languages to discard comment information.

1.4.1 Targeted Contributions

By the completion of these research objectives (RO), this research aims to contribute to knowledge in the following ways:

Contribution 1

Concerning IPT syntax feedback *and* semantic feedback having to be built very specifically for a given programming language:

We aim to "propose, described and test a state of the art method of providing guaranteed syntax feedback and semantic feedback in IPT, for theoretically every programming language".

The success of this contribution is assessed through RO1 (RO1a and RO1b). This is assessed through a technical analysis on the first language-independent IPT that was constructed as a product of this research. Through RO1a, we verify its "guaranteed" success rate. Through RO1b, we identify any limitations on working with "theoretically every programming language", and identify the contexts for those limitations.

Contribution 2

Concerning IPT having authoring tools needing to be made specifically for both the formatting of task creation, and the given language:

We aim to "facilitate task creation through an authoring tool that is capable of working across programming languages".

This contribution concerns tasks with one or more explicit solution *strategies* [32] (discussed further in section 2.2), and hence have an accurate method of gauging the success of tasks. Semantic feedback would be the result of conveying how close to success the user is to a solution strategy.

The questions produced for RO1 construct such an authoring tool. The contribution can hence be considered achieved if semantic feedback can be produced in tasks made in an authoring tool "that is capable of working across programming languages". The semantic feedback produced, should have feedback guaranteed to be produced, and should be accurate.

Contribution 3

Concerning finding a renowned repository that can be used for program synthesis[11] (as program synthesis was only domain-specific to programming languages):

We aim to "record the effects of using ANTLR4 grammar terms on the legibility of syntax and semantic feedback, in a language independent IPT".

ANTLR4 was the compiler generator chosen for this research. Reasons for this are further discussed in section 2.3, where we discuss compiler generator traits in further detail.

This contribution is evaluated through RO2 (RO2a and RO2b). RO2a is used to discuss whether it would be simple to identify, download and use grammars that one is unfamiliar with. RO3b assesses how commonly unfamiliar grammars parse code comments. We evaluate the success of using a "repository that can be used for program synthesis", by the success of using non modified samples from that repository.

RO2b deeply discusses the use of grammar token names. We "record the effects of using ANTLR4 grammars terms" to identify paradigms on what affects the legibility of semantic

[11] Program synthesis is the process of identifying a program within a programming language, grammar or high level specification.

feedback. We then quantify the frequency of those patterns among the ANTLR4 grammar repository. Either of these had yet to be assessed in any work.

The discussion in RO2b is performed for ANTLR4. However, discussion on identified paradigms may also be applied to any potential discussion on tokens written in a Backus Naur Format[12] - were the conditions to be replicated.

RO3a is used to provide a rough speed comparison to provide knowledge for future studies - namely any speed limitations.

Though this research chooses ANTLR4 as the compiler generator, aspects of methodology using it can exist for other compiler generators (parser generator, lexer generator or binary protocol parser) and their respective strengths and weaknesses[48][49].

[12]Backus Naur Format and other variants are discussed in Section 2.3

1.5 Organisation of the Thesis

- The theory used to support each of these is discussed in Chapter 2.
- The methodology of each of these is discussed more thoroughly in Chapter 3.
- The results obtained to aid in answering these and the answers to each question are in Chapter 4.
- A summary of the contributions of this research, and the final conclusion and evaluation are in Chapter 5.

Additionally, this thesis contains a glossary of terms used after Chapter 5.

1.6 Chapter Summary

Through section 1.1, we introduced the field and defined Intelligent Tutoring Systems (ITS), and stated how some ITS have been created and used to support teaching in computer science. Intelligent Programming Tutors (IPT) were defined as ITS for computer science and programming related courses. The lack of integration of research in the field of IPT was noted - causing domain-specific IPT (and programming language-specific IPT).

The following Section 1.2 also discussed another critical area with a lack of integration - pedagogy and pedagogical discussion. This is followed by a section identifying the one form of pedagogy implemented in IPT as part of its definition: Adaptive feedback in section 1.2.1. Adaptive feedback in IPT currently has one work that can return feedback regardless of the state of the learner's code as far as this research is aware. This IPT is known as the Intelligent Teaching Assistant for Programming (ITAP). As each work in the field is programming language specific, the work is limited to Python.

Following this is a section on an area of pedagogy explored by only one IPT study: Self-explanation in section 1.2.2. The one work, however, does this process extraneous to the IPT. Our research hypothesises that one can elicit the same benefits as self-explanation by encouraging learners to use code comments for summarising what they're doing. Additionally, the location of code comments could be read by the IPT - enabling it to know *when* to encourage code comments. However, the pursuit of investigating this may end up being language specific like the majority of IPT research.

The reoccurring issue we have identified has been the language-specific nature of IPT research. In section 1.3, we discuss this research's intended approach for a language-independent IPT. Furthermore, this enables investigating the use of adaptive feedback techniques (such as the ones used in ITAP) in theoretically all possible programming languages. Additionally, our research can approach investigating the use of code comments to elicit self-explanation for theoretically all possible programming languages, given the language has code comments and is defined.

In that section (1.3), we discussed a critical tool for creating an IPT by referring to code by abstraction. The tool is a compiler generator called ANTLR4.

Finally, we reiterate our aims and objectives in section 1.4. Our contributions are in creating the first language-independent IPT framework and reusable methods. Furthermore, we establish the information needed to know the restrictions in facilitating self-explanation through code comments for all programming languages.

Chapter 2

Research Background

Through this section, we discuss literature reviews the current author constructed to inform their research. First, we performed a literature review on secondary studies related to IPT to identify and list what known paradigms exist for IPT or related synonyms. This investigation was also conducted to identify any IPT that could also be defined as a cognitive tutor. Also, the research background study is presented in this chapter. For the related works section, we first we discuss ChatGPT to distinguish the the focuses, strengths and weaknesses of our research field (IPT) from Natural Language Processing (NLP) tools. We then finally discuss two IPT with adaptive feedback task activity frameworks, their contributions and their limitations.

2.1 Review of IPT Secondary Studies - is there a lack of pedagogical discussion?

ITS for programming is arguably inconsistently defined, with few primary sources of research acknowledging each other. It was possible that empirical research on IPT had unanimously used methods that were based on pedagogical research. For instance, it was possible that student models of IPT utilised "open learner models" (OLM) [50] (which could also be named as the "interaction" component of explainable AI [51]), or at least one method utilised an OLM in a "planning domain definition language" (PDDL) [52, 53]. It was conversely possible that the research did not discuss pedagogy in IPT and consisted of research that did not acknowledge the contributions of other research in the field.

To investigate what paradigms have been formed in the field, we performed a literature review on secondary studies (surveys, reviews, meta-analysis) on IPT-related synonyms or studies that contained IPT.

Ultimately, there were questions this research pursued to answer and identify as paradigms; to define the observed knowledge and gaps in the field. The questions to explore can be summarised as follows in Table 2.1. 'In IPT':

	Literature Review Aim - Find out:	Literature Review Objective - Performed by:	
1	'What techniques/achievements are only possible for (or relevant to) a given programming language?'	Summarising and listing -when secondary studies refer to programming, -whether they observe, list or discuss the programming languages used by IPT, -and whether they observe if any IPT support multiple programming languages. This was performed through reviews (secondary studies) of IPT.	
2	'What other techniques/achievements in general have acknowledged or been implemented alongside each other?'	Sumarising and listing -what synonyms and definitions they use for IPT, -what they are primarily making a paradigm of between IPT (their research aims), -and whether they observe, list or discuss when IPT utlise the same observed methods This was also performed through reviews (secondary studies) of IPT.	
3	'Would "quality of learning" be impacted by techniques/achievements being used in conjunction with each other?'	Splitting into 3 further Research Aims:	3 further Research Objectives
	a	'How is quality of learning defined in observations of the field (secondary studies)'	which can be reported as a list of educational psychology (pedagogical) concepts observed, and a list of paradigms observed.
	b	'What *is ideal* to be observed to determine quality of learning (or different aspects of the quality of learning)'	which applies both to how IPT react to learning, and how primary studies evaluate the success of their IPT. This would be two separate sets of information. First, a list of aims, research methodology and evaluation methodology corresponding to each definition of "quality of learning". Second, can be split into the next objective, referring to observations on the quality of learning used by the IPT itself:
	c	'How do various IPT account for quality of learning, and what do they not account for with quality of learning'	This can be partly or entirely answered in a list of documented methods on how learning is modelled in the IPT, where each documented method corresponds to each definition of "quality of learning".
x	Additional information	Finding -Journals each respective secondary study has been published to, -The year range of the used samples, -and the number of samples.	

Table 2.1 A breakdown of the literature review aims and objectives (talks to perform to answer the given aim)

The former two questions (1 & 2) refer to summarising and listing what is possible and has been achieved within the field, which could be performed through reviews (secondary studies) of IPT. To answer the latter of these questions (3), first, one would have to define how one determines "quality of learning", define what a decline or improvement to the quality of learning is, and ultimately define how one could evaluate (qualitatively and quantitatively) the quality of learning.

The disparity in how "quality of learning" is observed through IPT primary and secondary studies raise critical aims and objectives listed in 3.

This literature review was motivated by our pursuit to find ways that IPT methods could be integrated, but also find any IPT capable of making decisions based on a changing cognitive model ("student model") of the learner. There's a term specifically for ITS that attempt to observe the learner's cognition and decide actions accordingly. These ITS are known as "cognitive tutors". Cognitive tutors is a widely accepted term, but (at face value) there appears to be a lack in IPT - to form a cognitive programming tutor. Our research pursued finding and listing existing cognitive tutors and their innovations, how IPT discuss pedagogy, or what potential ITS methods exist in cognitive tutors that could be performed in IPT.

As it is inconclusive whether information on cognitive tutors is in IPT, my research pursued listing instances where pedagogy is discussed in secondary studies observing IPT. There are also a few meta-discussions on what ITS innovations that have yet to be applied in IPT, a few discussions to form a compilation of explored patterns, and a few comparisons of defined paradigms in IPT. There is at least one notable discussion between two IPT studies on adaptive feedback, where Keuning [25] discusses the observations of LeNguyen [1] relative to their own near the end of Keuning's paper.

The aims of my secondary study literature review included discussing various definitions of the field, and discuss various patterns observed in the field. The literature review also lists stated contributions within the field, and compile a list of classifications across the field. Finally, the study lists current problems that can be tackled within the field. Although a very recent study by Mousavinasab et al [11] also attempts to evaluate various paradigms in the field of ITS in general, our study is the first to attempt to standardise terms and discussions across secondary studies on IPT. Our study will be the first to discuss existing approaches to forming cognitive tutors within IPT (whether they exist, how they model learning, and what theories surround their approach).

Hence to investigate each of these questions, I performed a systematic survey of secondary study on IPT, as well as a systematic survey of secondary study on ITS that included our definition of IPT. This literature review primarily looked for what their aims and evaluations

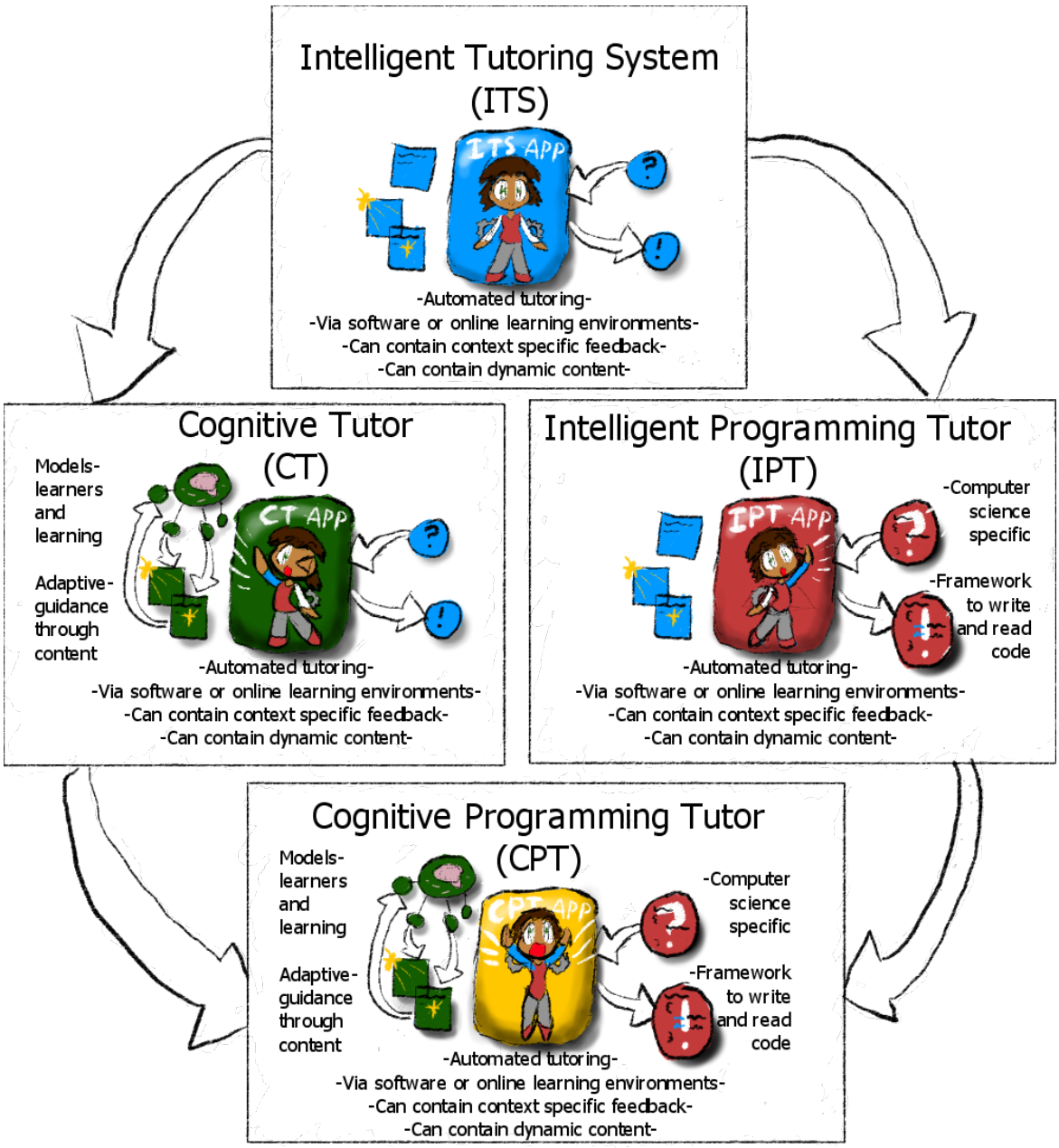

Fig. 2.1 An illustration of the shared components between "ITS", "IPT", "CT" and "CPT"

are, as well as what keywords and patterns are defined for their aims. Among these, we look for discussions on how the following papers:

- model learning,
- observed overlaps in primary research variables,
- languages facilitated by IPT,
- and how either the paper defines learning or what they observe of definitions of learning.

For any instance of secondary study research that includes programming but does not focus on programming, we also list how expansively or restrictively the focus of such a secondary study notes programming or computer science for ITS. How we observe and list each of these and what we include and exclude are discussed in Section 2.1.2.

Furthermore, this chapter also provides a concrete basis for the claims to the gaps in the field made in the subsequent Section 2.1.

Through this section, I document a literature review and discussion of our results for each of the detailed literature review aims.

To reiterate, the aims of this section are to find:

- Aim 1: 'What techniques/achievements are only possible for (or relevant to) a given programming language?'
- Aim 2: 'What other techniques/achievements in general are mutually exclusive?',
- Aim 3: 'Would "quality of learning" be impacted by techniques/achievements being used in conjunction with each other?',
  - Aim 3a: 'How is quality of learning defined in observations of the field (secondary studies)'
  - Aim 3b: 'What *(*is ideal) to be observed to determine quality of learning (or different aspects of the quality of learning)'
  - Aim 3c: 'How do various IPT account for quality of learning, and what do they not account for with quality of learning'.

A retrospective limitation of this literature review is due to the fact the results were obtained manually by one individual. A number of works listed in this literature review improved the integrity of the results by having multiple people conduct their sample-obtaining

process and then compare what they recorded for their paradigms. Additionally, a number of works obtained and studied their samples by using tools to automatically record data (through observed strings) into a database. This produced larger samples but would require each sample to be further checked to improve the integrity of their results. While the results of my literature review are not peer tested, the integrity of each result is confirmed by the current author.

2.1.1 Related Works

Keuning et al. [9] performed a study that contained "almost twenty" reviews of IPT. They found 3 papers dating before 2000, 9 papers dating between 2000 and 2009, and 6 papers dating between 2010 and 2014. [54] was the only paper included in their 2015. 12 of the papers were based on surveying learning tools rather than automated assessment tools.

2.1.2 Survey Method

To reiterate, for this particular search we aim to be able to list how various papers on IPT or IPT adjacent fields observe the state of the field relative to other IPT. This requires empirical papers with critical discussion on any subject for more than one IPT.

To compile a list of secondary studies on IPT to review, we used the strings in Table 2.2.

The list of papers was compiled through the Teesside university library search engine and ACM. The keywords searched are listed in Table 2.2. The reasons for each set of synonyms were as follows:

- For "Secondary Study Synonyms": As our research was on observations on IPT, the ideal papers to search for would be secondary studies, reviews, surveys and meta-analyses instead of individual primary studies on IPT. The "Tertiary" prompts were to observe if anyone attempted this form of study. "Tertiary" did not appear in any of the results. It is retrospectively worth noting that stating "Classif*" rather than "Classify" and "Classification" may have yielded the same results, as well as additional relevant results.

- For "ITS synonyms": Our research was on ITS, but the literal definition and how one can refer to the field can differ. We searched for terms that likely refer to the field.

- For "Subject synonyms": Out of ITS, we were specifically looking for IPT. We needed to specify the particular subject we were observing among each of the fields - programming, computer science or the act of "coding".

Search strings		
Secondary Study Synonyms	ITS synonyms	Subject synonyms
Secondary stud*	Intelligent tutor*	Program*
Review*	Cognitive tutor*	Computer science
Sensitive analy*	Adaptive tutor*	Code*
Literature Review*	Smart tutor*	Software Engineering
Review of the Literature	Adaptive feedback	
Literature Review*	Adaptive navigation	
Literature Stud*	Education* system*	
Map* Stud*		
Map* Review*		
Meta AND analysis		
State of the art		
Classify		
Classification		
Tertiary Stud*		
Tertiary Review*		

Table 2.2 The search terms used for this literature review. Every term in each set of synonyms is split by "OR"s, and each synonym set is split by "AND"s. A "*" causes search engines to include words that contain the start of that phrase

ACM digital library granted access to the majority of papers given through Teesside University.

The Teesside website yielded the remaining results, bringing the list of papers to check through our inclusion and exclusion criteria to 55.

In total, the initial search comprised 55 papers. Using the abstracts, this research further narrowed the results down to what was relevant by omitting the following:

- Papers that could not be accessed through Teesside University or through their previous versions.
- Primary studies or papers that strictly discuss 1 IPT.
- Studies not based on education.
- Studies without an explicit focus covering ITS (or a synonym of ITS).
- Studies on interactive systems *solely* to provide tests for a subject instead of containing any components that tutor a subject.
- Studies on subjects that are not related to computer science or programming.

A total of 15 of the 55 papers were included under this criteria. This method strictly used the 15 included papers. Further studies could include more modern papers (2022 and onwards), as well as include 'backwards snowballing' - the process of searching for relevant references cited by each paper and continuing until no more relevant citations pop up.

For each of the 15 papers, we read through and created a table on the following information:

The observation ratings were recorded as follows:

- Requirement (R) - Strict inclusion criteria, so any component on a given list would have had this component
- Observed (O) - Matches this observation to listed definitions, providing thorough detail
- Noted (N) - States if this component is present and lists the component, but it is not sorted or detailed
- Mutual (M) - Noted, but this component is mutual from the investigation or list. Likely used to inform another component
- Mutual Exclusion (ME) - Noted, but this component is factored only to exclude if it is defined in a certain way
- Unobserved (UO) - This component is not/sporadically discussed, not listed and has an explicit reason why it is not discussed.
- Not Observed (NO) - This component is not directly mentioned
- Not Applicable (NA) - A rating was not recorded

2.1.3 Literature Review

The results of the survey will be presented in this section. Through this section, we will present four tables of the results for the 15 languages, where each table is followed by a discussion.

Focus

Preceding discussion on the literature questions, we shall first discuss the variance in the research aims of these papers and the definition of the media they study. The full list of results can be viewed in Table 2.4-2.1.3.

Recorded	How it was observed
Year	Year of release
Study type	Whether the research discussed multiple primary studies of ITS (recorded as secondary) or a singular ITS (recorded as primary and excluded).
Subject	The mentioned or implied educational subject area of the study. The included subjects were 'general' (nonspecific), 'computer science' or 'programming'.
Media	A list of the terms used in the introduction for the media being surveyed, reviewed or analysed.
Level of Education	Whether a specified domain (higher education, further education, secondary education, K-12... etc.) is specified as an inclusion criterion of the paper or whether they include multiple, in the latter case, we give it an observation rating.
Focus	A summary of the particular topic this paper puts emphasis on discussing (often the keyword prevalent in their research aims).
Tutor Sample Count	The amount of included samples in their literature review. This is often mentioned at the end of their methodology and can be counted within their tables.
Date Range	The allowed year range of literature included (such as mine being 2015-2021). This could often be found in their inclusion/exclusion criteria. This was sometimes unstated, where the last date would be recorded as the release year.
Synonym for ITS or IPT	Out of the media included, we filled this with the phrase used for the one synonymous with ITS or IPT.
Computer Science inclusion	This is given an observation rating based on how they note whether their samples are based in Computer Science or Programming. For cases that are not a "requirement" that record the exact number, we give a percentage.
Definition of IPT	This is a quote of the phrase used to describe their Synonym for ITS or IPT. In some cases, the term was used with no definition or reference. We stated "assumed knowledge" in those instances.
Research Objectives	This is a quotation of the list of questions the paper proposes to answer through their secondary study (often at the end of their introduction).
Question 1 - Programming Language Limit	I gave an observation rating based on whether there's a mention of the used programming languages, a discussion, or a full listing of each sample.
Question 2 - Mutual research	Whether there was a discussion on observations on overlapping research themes or assigned paradigm values, I gave an observation rating based on how they discussed or recorded this.
Question 3a - QOL definitions	How the research discusses/observes how their samples define the pedagogy used to inform their methods. I gave an observation rating based on how this is recorded.
Question 3b - QOL observation methods	An observation rating is given according to how the research discusses/observes how their samples critically discuss successful learning results.
Question 3c - QOL observed by the IPT	Whether the research discusses or observes how each IPT assesses/calculates success. An observation rating is given based on how this is recorded.

Table 2.3

10/15 of the works stated empirical aims to answer. 13/15 stated their contribution. 6/15 of these papers had aimed to discuss feedback. 4/15 contains aims concerning the "authoring tool", with 2 explicit overlaps with a feedback aim. 3/15 of the works attempt to evaluate the effectiveness of (their definition of) ITS - also redefining the term.

One unique investigation was on sensors used to support the 'learner/student model' in Augmenting the Senses: A Review on Sensor-Based Learning Support" [55]; however, there was no incidental inclusion of any computer science or programming-related work, reflecting on a lack of use of any form of sensors in IPT. This work included 55 works for education.

Another unique investigation was on a specific form of 'learner model', the OLM in [50]. A lack of identification of any computer science implies a lack of OLMs in computer science, even among the 114 included works.

These two works are included, as inclusion criteria of the above two works would include computer science or programming-related works were they to encounter any, despite both having more than 50 works. Hence, both sensors and OLMs are arguably unobserved in IPT. Whereas sensors require specific hardware, OLMs are applicable to any ITS with a learner model - allowing their inclusion to be domain-independent given a learner model.

10/15 of the works explicitly used the phrase "ITS". The terms "AI", "adaptive", and an implication of a "domain model" or "expert model" occur in the following works. In [56] Dermeval et al. mention "adaptive tutoring" and imply the use of an 'expert domain model'. The tools are summarised as using "AI". In "Effectiveness of Intelligent Tutoring Systems: A Meta-Analytic Review" [13], Kulik et al. state they are "Grounded in [AI] concepts" and mention "expert-knowledge database". In contrast, two works define ITS with the addition of a 'pedagogical model' and 'learner model'. In "Systematic review of research on artificial intelligence applications in higher education - where are the educators?" [27] Zawacki-Richter et al. mention the "learner model", and have their paper focus on pedagogy. In "Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods" [11] Mousavinasab et al. explicitly list the 4 model types that commonly appear in ITS primary studies ("expert/domain model", "learner/student model", "interactive model" and "pedagogical model"). Both of these papers critically discuss the semantics of pedagogy and what AI means in ITS, respectively. In all other works, they assume the use of the term ITS is known and provide no definition.

The unanimous components of ITS are, being an AI, containing a domain model, and containing adaptive feedback. In addition to this, (by Kulik's definition), for it to be an ITS rather than a "computer aided instruction" (CAI), it must engage with the learner as they work on the problem, and not *just* after they reach a solution. The addition of a "pedagogical

model" and a "learner/student model", mentioned and discussed by Zawacki-Richter and Mousavinasab, might have a preferable distinction as Cognitive Tutors.

Tang el al.'s work [12] mentioned ITS as a potential surveyed "AI-supported-e-learning" but did not define ITS. Surprisingly, none of the key papers noted by Tang et al's discussion on AIeL research networks [12] occurred in our bibliography. This may imply a lack of overlap between AIeL trends and our definition of ITS, a minority of ITS in their study, or that our research's focus themes differed considerably from most ITS research.

Paper Ref	Focus	Date Range	Sample Count	Term for ITS or IPT	Computer Science	Definition of IPT	Research Question(s) or Aims	Q1	Q2	Q3a	Q3b	Q3c
[1]	Adaptive Feedback	1999-2010	20	1-Educational systems for programming with "adaptive feedback"	Requirement	"That is, adaptive feedback is provided differently for individual students by analysing the student's action (e.g., student's solution attempt) and/or adaptive feedback may be provided by an education system that is based on a student model."	This paper identifies analysis approaches for programs and introduces a classification for adaptive feedback supported by educational systems for programming. The classification of feedback is the contribution of this paper.	NO	O	N	N	O
[57]	Adaptive Feedback	2000-2016	20	1-Adaptive Learning Environments 2-"keywords such as feedback, adaptive feedback, intelligent tutoring system, adaptive learning system, computer-based tutor, pedagogical agents, and computer-assisted learning were used."	Observed (30%)	"Adaptive learning environments provide personalization of the instruction process based on different parameters such as sequence and difficulty of task, type and time of feedback, learning pace, and others (Brusilovsky et al., 1999; Stoyanov & Kirchner, 2004)."	"to compare computer-based learning environments according to their implementation of feedback and to identify major open research questions in adaptive feedback implementations" "this study reviews various implementation of feedback, based on the four adaptation characteristics: means, target, goal, and strategy (M. E. Specht, 1998)"	UO	O	O	O	O
[9]	Categorising Adaptive Feedback	x-2016	55 + 19AA	1-learning environments for programming 2-automated assessment (AA) tools	Requirement	learning programming, mostly on learning environments for programming or automated assessment (AA) tools	"What is the nature of the feedback, how is it generated, can a teacher adapt the feedback, and what can we say about its quality and effect?"	O	NO	O	O	O
[55]	Categorising Sensors in assessments	2012-2014	79 (55 Educational)	N/A	None	N/A	-Learning domains: get an overview of sensors and learning. -Formative assessment: focus our research in sensors and Learning, exploring how they can assist with a main current educational challenge. -Feedback: deepening our research in sensors and Learning studying how they have been used for giving Feedback, which is a key element for Formative assessment and one of the most important interventions in learning.	NA	NO	O	O	O
[56]	Authoring tools	2009-2016	33	Intelligent Tutoring Systems (ITS)	Not Observed	"Intelligent Tutoring Systems (ITSs) are concerned with the use of artificial intelligence techniques for performing adaptive tutoring to learners according to what they know about the domain (Sleeman and Brown 1982)."	which ITS components can be authored?; which ITS types can be designed by authoring tools?; which features facilitate the ITS authoring process?; what authoring technologies have been used to design ITS?; when does the authoring occur?; what is the evidence that support reported benefits of using ITS authoring tools?	NA	O	N	O	O
[58]	Authoring Tools	N/A	8	Intelligent Tutoring Systems (ITS)	Not Observed	N/A	Searching and categorising ITS Authoring tool types	NA	N	NO	NO	NO
[25]	Adaptive Feedback (Thoroughly)	x-2015	101	Automated Feedback in Programming Exercises Automated Assessment Tools (AAT) Intelligent Tutoring Systems (ITS)	Observed	ITS (Assumes known)	"What is the nature of the feedback that is generated?" "Which techniques are used to generate the feedback?" "How can the tool be adapted by teachers, to create exercises and to influence the feedback?" "What is known about the quality and effectiveness of the feedback or tool?"	O	O	N	O	O
[13]	Changes ITS invokes to test results across Primary Studies	NA	50	CAI (Computer Aided Instruction Tutors) - 1st generation ITS (Intelligent Tutoring Systems) - 2nd generation	Mutual	Full section discussion the differences between CAITs and ITSs: "does the computer tutor, like a human tutor, help learners while they are working on a problem and not just after they have recorded their solutions?"	How effective are ITSs? Do they raise student performance a great deal, a moderate amount, a small amount, or not at all? If ITSs do have positive effects, has their effectiveness declined with the fine-tuning of the systems in recent years? What accounts for the striking differences in review conclusions about ITS effectiveness?	NA	O	NO	NO	NO

Table 2.4 The first half of a summary of contents in secondary studies on IPT

Paper Ref	Focus	Date Range	Sample Count	Term for ITS or IPT	Computer Science	Definition of IPT	Research Question(s) or Aims	Q1	Q2	Q3a	Q3b	Q3c
[10]	Programming based ITS	NA	40	ITS for programming education	Requirement	A number of intelligent tutoring systems have been created for programming education. For the purposes of this review they will be referred to as Intelligent Programming Tutors (IPTs). IPTs are a distinguishable subfield within intelligent tutoring systems as learning programming poses problems that are not found in other areas.	(1) What programming languages are being taught with IPTs? (2) What types of supplementary features are used in IPTs and how have they been implemented into tutoring tools? (3) What parts of the tutoring process are adaptive within IPTs?	O	N	NO	NO	NO
[50]	"Open Learner Model (OLM)" "Learner Analytics Dashboard" (LAD)	NA	140	Intelligent Tutoring Systems (ITS)	Not Observed	ITS (Assumes Known)	1. What data is collected in OLM systems, and what type of modeling methods are used? 2. What are the current trends in OLM research in terms of publication venue, publications over time, authors, and top cited articles? 3. What are the central themes or topics that have emerged from OLM research articles? 4. What is the nature of OLM system evaluations? 5. What similarities and differences exist between OLMs and learning analytics dashboards?	NO	O	U	O	O
[27]	Studies (qualitative or quantitative)	2007-2018	150	Intelligent Tutoring Systems (ITS)	Observed (42%)	Intelligent tutoring systems (ITS) can be used to simulate one-to-one personal tutoring. Based on learner models, algorithms and neural networks, they can make decisions about the learning path of an individual student and the content to select, provide cognitive scaffolding and help, to engage the student in dialogue	- How have publications on AI in higher education developed over time, in which journals are they published, and where are they coming from in terms of geographical distribution and the author's disciplinary affiliations? - How is AI in education conceptualised and what kind of ethical implications, challenges and risks are considered? - What is the nature and scope of AI applications in the context of higher education?	NO	N	O	O	O
[11]	What does the "AI" mean in E-learning and ITS	2007-2017	53	Intelligent Tutoring Systems (ITS)	Observed (38%)	The ITSs have a classical architecture with four modules which are known by different names in studies (Then lists the 4 models)	RQ 1. For which educational fields ITSs have been designed? RQ 2: Which AI techniques have been applied in the development of ITSs? RQ 3: What are the main purposes of using the AI techniques in ITSs? RQ 4: Which factors have been used for representing the adaptive or one-to-one instruction in ITSs? RQ 5: What types of user-interface have been used for development of ITSs? RQ 6: Which methods have been employed for the evaluation of ITSs?	NO	N	O	O	O
[12]	Research trends and citations	1998-2019	105	Intelligent Tutoring Systems (ITS)	Noted	ITS	- Which were the top 10 countries and authors integrating AIeL in the field between 1998 and 2019? - Which major journals published AIeL studies between 1998 and 2019? - What were the research methods and application domains adopted in AIeL studies from 1998 to 2019? - What were the roles of AI in e-learning? What were the most frequently cross-referenced research streams for AIeL between 1998 and 2019? - To what extent were these streams extended by follow-up papers? In addition, what is the visualized structure of the main AIeL literature from the perspective of these papers?	NO	O	NO	N	NO
[28]	Agents used alongisde learners			Intelligent Tutoring Systems (ITS)	Not Observed	ITS	The primary objective of this review is to provide a critical analysis of the effectiveness of ABLEs in facilitating students' experience of adaptive emotions.	NO	NO	N	N	O
[31]	When learning confusion benefits	N/A	N/A	N/A	N/A			NA	NA	N	N	NA

Table 2.5 The second half of a summary of contents in secondary studies on IPT

Programming Languages

This section discusses our first literature review aim. To reiterate, this research wanted to find how secondary studies discussed "what techniques/achievements are only possible for (or relevant to) a given programming language?"

In this section, we first discuss the number of reviews that include programming and computer science ITS before discussing whether works observe the used programming languages. This allows us to discuss how applicable the observations are to specifically computer science and programming. The results are summarised in each table in this subsection, with Table 2.4-2.1.3 having the rows relevant to this section.

8/15 of the works mentioned if works were for contained computer science or programming. 4 of these *required* all works within to be for computer science or programming. [1, 9, 10, 25] were strictly based on programming exercises.

Out of the remaining works that include but do not require works based in computer science or programming, 3/5 of them give exact values of the number of IPTs included. "Adaptive feedback in computer-based learning environments: a review" [57] has 30% of works in computer science or programming. [27] has the majority 42% of works in computer science or programming. [11] has the majority 38% of works in computer science or programming.

Ultimately, only 3 of the papers could be counted to have 'observed' the programming languages used by their samples. These were [59, 25, 10]. The only work on Computer Science and programming that did not contain a list of programming languages was [1].

Keuning et al.'s work [25] noted 12 studies that contained work that supported (or has been applied to) more than one programming language ("11.9"%). However, they also note, "The remaining tools support multiple languages of different types and paradigms and are often test-based AA systems". The other works that noted programming languages only noted their works supporting one language. "Migration is applied in INTELLITUTOR, which uses the abstract language AL for internal representation. Pascal and C programs are translated into AL to eliminate language-specific details. After that, the system performs some normalisations on the AL-code", however, Intellitutor [60] is not an IPT, but an AAT.

All other papers were either recorded as either Unobserved, Not Observed or irrelevant to the question, as their focus was not on programming or computer science and hence did not focus on particular aspects unique to programming ITS.

From this, we can answer that at least up to 2016, IPT supporting more than 1 programming language is minimal, and no IPT is programming language-independent in implementation.

As for observations on language-specific trends in IPT, Keuning et al. [9, 25], and Crow et al. [10] note domain-specific qualities of ITS to programming. These papers both also include programming languages to "contextualise" features prevalent in each tool. However, there is no listed paradigm that notes a difference between programming languages or types. Keuning separates the languages between "Imperitive/object-oriented" ("73.3%"), "Logic" ("7.9%"), "Functional" ("5.9%") and "Unknown" ("1%") and discusses them as trends only.

Hence, out of all research on the field, it's inconclusive what IPT paradigms are influenced by language-specific features.

Investigated Mutual Research

Our second literature review aim concerned finding what secondary studies considered, "What other techniques/achievements in general are mutually exclusive?"

Through this section, we will discuss the findings of each secondary study on research and techniques performed with each other relative to their research focus.

11/15 of the secondary studies at least note findings on what techniques/specifications are met in conjunction with each other, with 7 of those 11 listing so empirically by a paradigm.

Firstly, the two works have a figure discussing any used adaptive feedback methods that overlap. [1] lists the number of studies with feedback types that occur in conjunction with one another. The feedback types were on/off, syntactic, semantic, quality and layout. There were 5 paired feedback type uses identified out of the 20 sampled IPTs. [57] provides a Venn diagram between Adaptive Feedback Means, Adaptive Feedback Target, Adaptive Feedback Strategy and Adaptive Feedback Goal. The majority of works contained each, and all but one work contained at least 2. However, what type of Adaptive Feedback each category contained differed between IPT. In [25], the returned feedback types are critically discussed and presented in a bar graph against year.

In [56] Dermeval et al. state when discussing the different types of "ITS Types" stated, "[a] result that deserves some attention is that almost 30% of the papers are categorized as Non-specific. This result may indicate that there is not a shared understanding in the ITS community of the underlying theories, technologies, and features of ITSs since many researchers are developing authoring tools for designing their own type of tutor."

[25] contained a graph on overlapping implementations of domain-specific (features of ITS - and AA - unique to the field of computer science or programming). These were "Dynamic code analysis using automated testing (AT)", "Basic static analysis (BSA)", "Program transformations (PT)", "Intention-based diagnosis (IBD)", and "External tools (EX)". Among the domain-specific combinations, only AT+BSA (30-81.1%), AT+PT (16), IBD+PT (16-42.1%), AT+EX (12-100%) and PT+BSA (11-29.7%) occurred more than 5

times. IBD+AT (<28.6%), IBD+BSA (<28.6%), EX+BSA (<41.7%) and EX+PT (<41.7%) occurred in the same IPT/AA less than 6 times.

Two works contain explicit aims to compare the components of the works they sample to a synonymous term. [50] contains the research aim, "what similarities and differences exist between OLMs and learning analytics dashboards?". This research contains an additional section on "RECOMMENDATIONS TO MERGE OLM AND LAD RESEARCH FIELDS". [13] also discusses past and modern ITS, comparing CAI.

Finally, "Trends in artificial intelligence-supported e-learning: a systematic review and co-citation network analysis"[12] attempts to identify research trends in the field - who gets the most citations and what research follows. However, this work does not discuss the content of the ITS. It observes shared patterns in the primary studies surrounding the ITS.

Phrases implying a lack of integration in the field were common among all works marked noted or observed for question 2. Tang et al. attempted to find insight into research trends to find what researchers focus on, but this does not give conclusive insight into the problems that prevent research techniques from being implemented alongside each work. Each of the specialist studies provides insight into what IPTs share components and what components are lacking. A strong implication of the problem is how inconsistently defined the field is. For instance, "IPT" is not a unanimous term.

To answer from these studies, "what other techniques/achievements in general are mutually exclusive," this depends on the focus of the question and is answered by the plethora of paradigms on each subject. Even when discussing feedback, it can be split between output structure [1] to what methods are used to form the feedback [25]. However, from the recurring statement and aims of finding shared research and terms in the field, we could arguably say research in the field is currently non-integrated.

Definitions of Success/Educational Psychology in ITS

In this section we will discuss the 3 questions concerning the quality of learning in sequence against each of our surveyed secondary studies.

For "What *(*is ideal) to be observed to determine quality of learning (or different aspects of the quality of learning)", we list for each secondary study how they note any information on how their samples observe learning. We discuss any collective patterns that occur from this.

Concerning "How is quality of learning defined in observations of the field (secondary studies)?",

We found that most papers observed this in ITS by classifying the form of research being conducted[1]. If the paper discusses the explicit method used for obtaining data on their quality of learning, we record that as "observed".

Out of the works that observed methods in the works they sampled, 8/10 also provided an answer to "How do various IPT account for quality of learning, and what do they not account for with quality of learning".

Schneider et al.[55] provides unique observations on the quality of learning in their samples. They discuss the use of sensors to evaluate learning and support observations of learning in ITS.

Certain ITS observe the quality of learning as one component, such as quantitative test scores in Kulik et al.'s work [13]. In such cases, quality of learning resultantly gets categorised as "Not Observed" in each category. While Kulik's work defines the quality of learning in their work, they do not discuss the pedagogy of other works. They also exclude works that do not provide specific quantitative results (standardised test scores). Tang et al.'s work [12] does not study the ITS within their primary study samples, but the research trends of those works. They discuss the type of research being performed but not the contents within (aside from the titular theme of AI in e-learning). Hence the quality of learning is not discussed outside of research types.

Another trend among the results was for the 3 out of 4 papers discussing feedback. As they discussed an aspect of pedagogy (feedback), they defined learning relative to its impact, often quoting constructivism. They did not have a paradigm for noting how learning is defined but would critically discuss how each primary study observes learning.[1, 57, 9, 25] each discuss feedback. The exception to this is Le et al.'s work [1], which only notes the type of empirical research performed.

Overall, it was overwhelmingly common for the secondary studies to list how each work's ITS observes learning (as the focus is on the ITS). This was the case in 10/15 studies. It was slightly less common for the secondary studies to list how each primary study evaluates learning. This was the case in 8/15 studies. Each work observed many primary studies discussing pedagogy, which implies pedagogical discussion is not lacking in works in the field, or observations of the field. However, only 1/3 of the works strictly including programming discussed pedagogy; [57]. They stated, "The most common techniques used for evaluating adaptive feedback implementations are through questionnaires, pre-test and post-test, and analysis of log data". Finally, it was even less common for secondary studies to discuss differences in how learning is observed. 9/15 of the works made some form of remark and 5/15 works provided paradigms.

[1]IE: "Empirical, Analytical, Anecdotal" [9], "Qualitative, Quantitative, Technical" [1]

2.2 Additional Paradigms - Related works

Through this section, we summarise various paradigms we acknowledge/use in this thesis. These paradigms come from secondary studies and primary studies of IPT. One such primary study is a state-of-the-art IPT we compare our research to.

This paper concerns the use of automated *tutoring* tools to *improve learning* in computer science and programming. Hence, we focus on IPT tools containing frameworks that provide feedback to improve learning. We discuss AAT if they are built into IPT, or in contexts, they are discussed alongside IPT.

Many IPTs and AATs have been constructed, with primary research surrounding most. There also exist several secondary studies defining IPTs and AATs in various ways based on numerous reoccurring patterns and unique traits among them. Through this section we discuss a number of these patterns, to provide context for common problems.

IPT facilitate programming based learning activities. Most have varying methods of returning feedback. IPT have different rules concerning how they determine if a learning activity has reached a solution. Nguyen-Thinh et al. produced a 5 class classification for how activity solutions can be explicitly defined[32]:

- Class 1 - One defined solution.
- Class 2 - One solution strategy, different ways of observing.
- Class 3 - Multiple limited solution strategies.
- Class 4 - Multiple solution strategies that may not all be listed/observed by the tutor at the time of creation, but can be verified to be correct.
- Class 5 - Problems with solutions that can not have their correctness verified automatically.

Nguyen-Thinh et al then evaluate implementations of 14 IPT against this classification [59], stating solutions up to class 3 exist. This paper 2014 existed before works such as the IPT ITAP documented by River's ITAP [16]. ITAP supported multiple solution strategies, but also had a unique contribution for verifying and adding solutions from other users (learners) to the same activities. This enables ITAP to consider solutions not originally defined by the tutor. ITAP arguably becomes the (proposed) first instance of Class 4 activities implemented in IPT. We will discuss ITAP further in related works. Ultimately, Class 1-3 and arguably 4 are currently present in IPT.

An empirical study by Keuning et al [25] noted and referenced 101 IPTs and AATs as of 2018. Their study discussed both primary studies of the works, and secondary studies

reviewing several works. They excluded works that did not teach programming for the sake of learning programming or computer science. They also excluded works containing only class 1 solution strategies, citing Nguyen's activity class classification [59]. Unlike this paper, they did not regard ITAP[16] as Class 4. Keuning et al [25] discuss numerous methods used in IPT and AAT, ranging from:

- "Feedback Types" with multiple sub categories (and the titular discussion),
- "Techniques" for comparing learner's answers with implemented solutions,
- "Adaptability" of feedback that can be specified by tutors, and
- "Quality" of the research pertaining to each IPT or AAT.

Another noted component of IPT is not just their ability to evaluate the correctness of solutions, but also to return *constructive* feedback. Without the latter (constructive feedback), they would only be AAT. IPT differ in how feedback is delivered.The type of feedback provided can also be critical to improving the motivation of the learners, or consequentially hindering the motivation of learners. Kyrilov's empirical study notes how instant "binary" feedback results in learners losing motivation [2]. Rivers also notes for ITAP [17] learners were more motivated to use a later model of ITAP that gave more detailed feedback on the problem. Alternatively, the noted improvement from having more detailed feedback could be from a higher likelihood of hints covering the information needed.

In the "more detailed feedback" versions of ITAP, it returned feedback in the following pattern with no ambiguity: [Location info] + [action verb 1] + [old value] + [action verb 2] + [new value] + [context] (Figure 30 in [17]) We further discussion of ITAP occurs through Section 3.1, where ITAP's implementation is discussed relative to our research's methodology.

Le Nguyen had another secondary study that classified feedback through IPT [1], and this paper was also discussed in Keuning et al.'s paper. Their study strictly included IPT with 'adaptive feedback', resulting with 20 IPT studied through this work. This included 3 binary feedback IPT, which were adaptive in the sense of content specifically made for the learners. "Since both QuizJET [61] and QuizPack [62] are able to generate parameterised questions, the systems provide adaptive feedback" and "Since M-PLAT [63] is an adaptive educational system as the authors claimed...the feedback messages are provided individually to students". Such feedback was classified as "Yes/No" feedback.

Nguyen also notes:

- "Syntax feedback" - remarking on compilation errors,

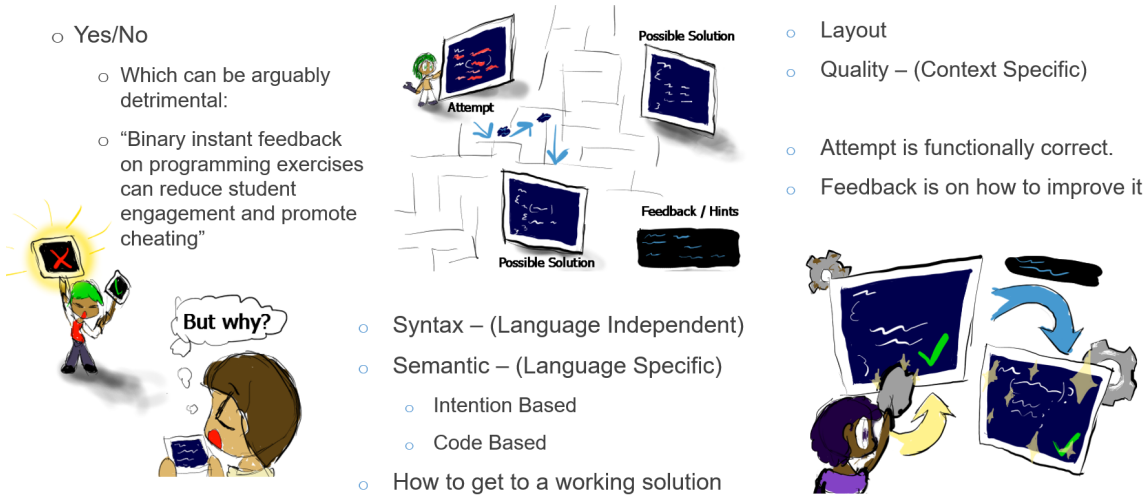

Fig. 2.2 A comical illustration on the 5 types of feedback defined by Nguyen [1]

- "Semantic feedback" - remarking on errors in regard to the learning activity. It is split between:
  - "Intention-Based Feedback" - Identifying what type of solution the learner is aiming for.
  - and "Code Level Feedback" - Identifying erroneous code in regard to the learning activity.

  , All cases of Semantic feedback contained code-level feedback, but not all cases contained Intention-based feedback.
- "Layout feedback" - remarking on corrections to the learner’s writing convention.
- "Quality feedback" - remarking on corrections to the efficiency of the code (such as measuring speed or memory usage).

We illustrate these 5 types of feedback in Figure 2.2.

2.3 Backus Naur Format

In this section, this research first defines a known convention for writing programming language grammars. First we define "Backus Naur Format" (BNF), discuss compiler generators that use this format, then present literature observations on the compiler generator this research uses.

BNF is a popular writing convention for defining the syntactic rules of a programming - a "meta-language" [64]. It's definition appears in works such as McCracken's [64] and Quinlan's [65].

BNF has been widely adopted, with several widely adopted variants. Quinlan et al [65] list and define the following variants:

- BNF - The original format
- EBNF - *Extended* BNF. Contains a few formatting differences, and facilitates the ability to handle "repetition of syntactic rules", "special sequences", "optional choice of syntactic rules" and "exceptions to syntactic rules".
- ABNF - *Augmented* BNF. "The differences between standard BNF and ABNF involve naming rules, repetition, alternatives, order-independence, and value ranges" [66]. However, overall contains no features that EBNF lacks.
- RBNF - *Routing* BNF. Contains the ability to write exception rules.
- LBNF - *Labelled* BNF. "A BNF grammar where every rule is given a label" [67]. Additionally adds "abstract syntax conventions, lexer rules, pragmas, and macros" [67] to BNF.
- TBNF - *Translational* BNF. Designed as an extension of EBNF that accommodates non context free components: structuring how the grammar will be read, and allowing instructional code.

BNF is predominantly for defining abstract syntax rules, however has also been applied for mathematical notations - particularly from the variant MBNF [65].

EBNF is regarded to be complete in its ability to define any context free grammar, and has hence been a widely adopted format. BNF and its variants are often used for compiler generators such as Lexx/Yacc (which is based on BNF) and ANTLR4 (which is based on EBNF, with the addition of ANTLR4 only syntax)[49]. An example of an ANTLR4 grammar is in figure 2.3.

Because BNF is shared among many compiler generators, research continues to developed practices on grammar writing, as well as develop practices for algorithms reading the BNF grammars. Compiler generators exhibit various differences ranging from:

- grammar writing convention (variants and additional features of BNF),
- algorithm used to parse grammars,

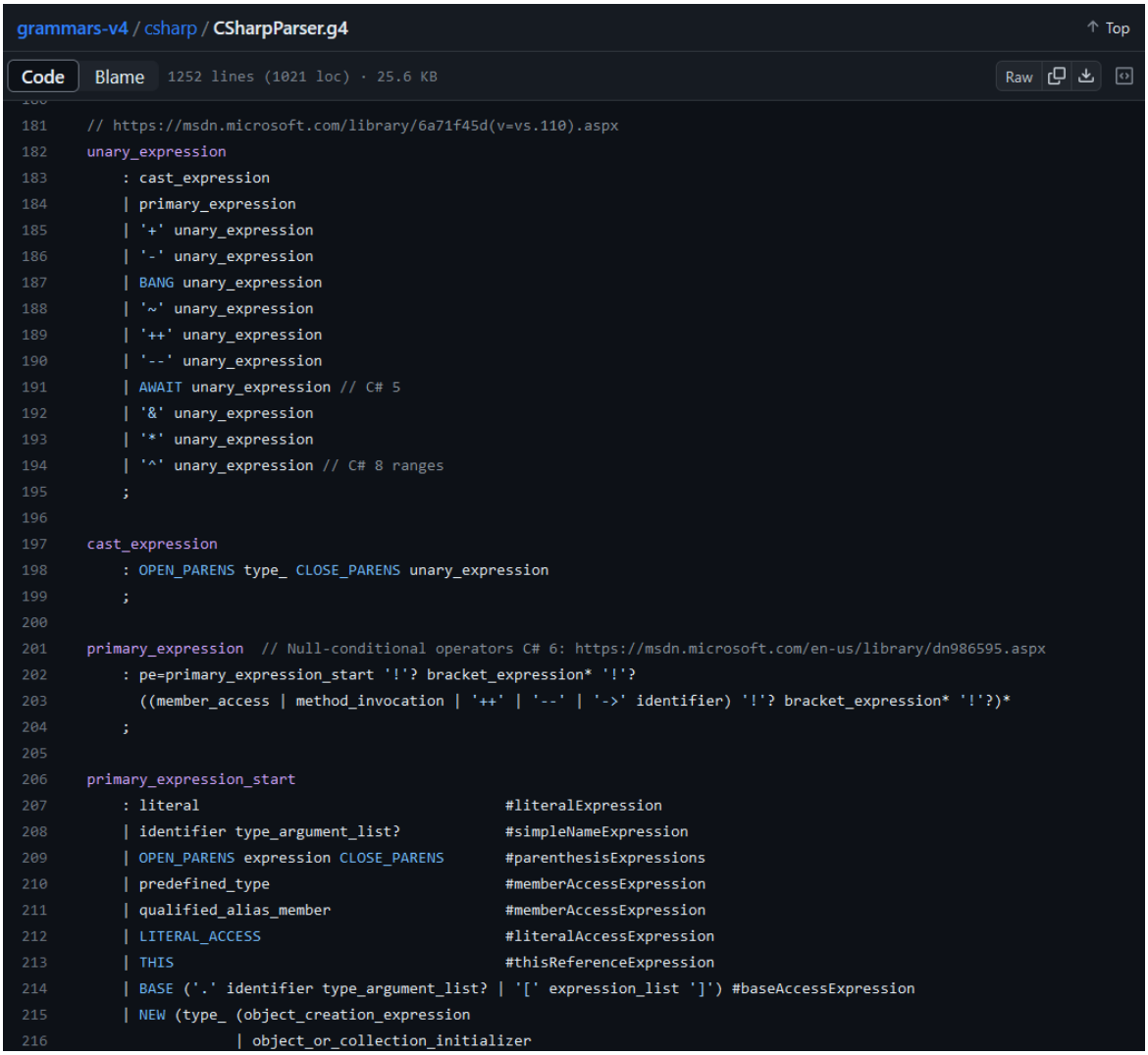

Fig. 2.3 The parser of a C# ANTLR4 grammar on the ANTLR Github repository. Resembles EBNF.

- containing repositories archiving grammars written for the compiler generator,
- programming languages the compiler grammars compile into,
- and what structures the compiler takes (and features they support) when compiled.

[48] They also differ in what standards they support. For instance, ANTLR4 facilitates context-free grammars, yet also facilitates the ability to write context specific actions into the grammar. Theoretically, methodology concerning writing grammars, compiling them and using them within tools could be applied across compiler generators. However, the output of these tools can be domain specific, which can inhibit compiler generators from being tested interchangeably. While it makes it difficult to test across compiler generators, our research believes any discussion on the phrasing of EBNF grammars can be applied to other compiler generators.

The chosen compiler generator of this research is ANTLR4. It contains the uniquely developed the "Adaptive LL"(ALL) algorithm for tokenising and parsing [46]. This algorithm is an adaptation of "LL(*)" to enable the parsing of grammars to identify ambiguities dynamically at parse time (rather than statically defined by the construction of the grammar). In practice, it is able to parse non-left-recursive context-free grammar.

ANTLR4 theoretically supports the creation of all possible programming language grammar definitions (given they are context-free and purely literal). Additionally all compilers generated by the ANTLR4 tool will still complete compilation, even when the code being compiled contains errors. This allows ANTLR4 to log syntax errors in the generated ASTs. This applies for all language grammars compiled with the ANTLR4 tool. This is critical for an IPT system with aims to identify and correct learner mistakes.

ANTLR4 additionally supports specifying action code, however this standard is discouraged in general use on sites such as the ANTLR4 repository. Tokenisers and parsers (forming a compiler) created by ANTLR4 facilitates two ways to iterate through code as it is compiled into an AST for each grammar.

First is the listener which allows action code to be programmed against each token as it is read.

Second is the walker, which also allows action code to be programmed around each token, but requires explicit calls to navigate an AST.

The generated compiler code can be written in a limited number of programming languages:

- Java,
- C#,

- C++,
- Python2,
- JavaScript,
- Python3,
- Swift and
- Go.

This research benefits from the output language being in C#, as it enables use within Unity software environment (discussed in Section 3.1).

In the generated compiler code, the listeners (and walkers) inherit from an abstract class used across ANTLR4 grammars. Using the ANTLR4 listener's abstract class, we theorised an IPT can dynamically construct its own AST structure as ANTLR4 constructs its own. ANTLR4 additionally contains a repository. It is widely supported with more than 200 languages. However, the repository is not as popular as other tool's repositories such as "Lex/Yacc" [49].The entries on the repository are also open source, primarily rely on user contributions and their adherence to newer[2] repository standards. Works such as Dean's [48] ranked grammar repositories of different tools, where ANTLR4's existed but was given the lowest rank in terms of "the comprehensiveness of the existing repository of grammars available online".

Ultimately, ANTLR4 was the chosen compiler generator for this research because of the following combination of traits:

- The current author was first familiar with ANTLR4 as a compiler generator compatible with all languages,
- ANTLR4 has arguable relative ease of use as a compiler generator[49],
- ANTLR4 was suitable for this research for the following purposes. It contains:
  - An improved parsing algorithm (ALL) and writing format (EBNF). This helps facilitate all possible literal programming languages[46].
  - An open source repository of its grammars which included well renowned programming languages.

[2]ANTLR formerly could output compilers in only Java, which would also lead to a lot of Java specific actions. Code of grammars has been easing into supporting more programming language

- The ability to build into C# - among other languages - where C# is relevant to how this project approached building an IPT.
- The ability to refer to compiler code by abstraction.

In the following section, we discuss works that use the ANTLR4 repository, and observe if there are any instances discussing the "legibility" of the strings used for the terms.

2.3.1 ANTLR4 grammar uses

For the definitions of grammars, This research will also discuss how the semantic differences affect the feedback provided by programming languages. Multiple grammars for a given language can be built differently. They can target different versions of ANTLR4, target different standards for writing grammar, or simply differ in how they're phrased and structured. The current ANTLR version as of this writing is 4 by Parr [46]. Additionally, the literature surrounding ANTLR4 [45] promotes "context free grammars"[3].

It is possible there is little research surrounding the phrasing of grammar because it's not a functional requirement in their usage. Our research aims to refer to the token names to provide structure context in feedback, as well as to tutor the names of the structure. To perform this in a multilingual IPT, our research must pull the names as defined from the grammar to provide feedback. This could affect our feedback in a way illustrated in Figure 2.4. As it is not a requirement of language grammar or used in their general output, there is a reasonable chance grammar does not account for how their phrasing would be observed by secondary or tertiary users. Through this section, we investigate this conjecture.

To find how ANTLR grammar semantics have affected existing research, this research performed a quick analysis of empirical studies using ANTLR. The ideal search of "ANTLR (OR ANTLR4) grammar (semantics OR phrasing) (survey OR "meta-analysis" OR review)" as titular (with no year restrictions) words yielded no results on Google Scholar.

"ANTLR (OR ANTLR4) grammar" as titular words yielded 16. 4 of those 16 were included for containing accessible web links and full documents, and due to not being a primary study on the creation of a grammar. Each was empirical [43, 68–70]. This search also contained no date restrictions for the papers, yet the range contained papers from (2005-2022).

Of the four works, Bovet's was a discussion (co-authored by the creator of ANTLR) on facilitating a tool for writing ANTLR grammars [70]. Semura's [43] precedes Zhu's [68] on implementing Code Clone detection for all programming languages ("Multilingual") using ANTLR4 [68]. Semura's sampled 150 languages on the ANTLR4 repository and excluded

[3]Programming language grammars that are designed to work for any given use of the ANTLR4 tool

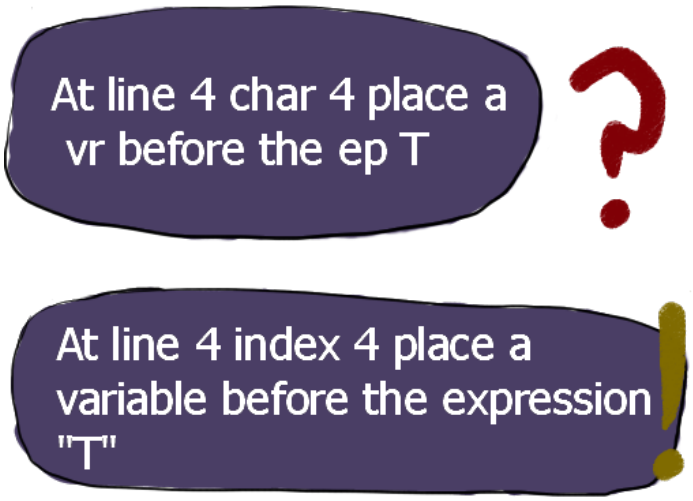

Fig. 2.4 An illustration on how grammar phrasing could potentially affect the phrasing in feedback.

"esoteric" languages, leaving 43. Zhu's work was sampled against 20, where each was successful, given no keyword restrictions (at the expense of information omissions). Wal's [69] covers research on converting ANTLR4 grammars to "(BNF)-like Grammar Format". It was sampled against all 370 ANTLR4 grammars on the repository (at the time). Only 6 did not compile by the ANTLR4 tool, preventing them from being tested with the extractor.

Although each noted how the phrasing of each grammar could affect the output, there was no observed discussion on the exact phrasing used within grammar. In Wal's research [69], as part of their first research question, they surveyed each ANTLR construct; the number of grammars containing them. However, they did not need to investigate how each construct could be phrased. Semura's work [43] noted 4 ways of writing comments in languages for the purpose of omitting; hence, it was not important for them to discuss the phrasing.

2.4 Related Works

Through this section we first discuss ChatGPT, a popular Natural Language Processing (NLP) tool that has been applied to programming and education. Secondly we discuss Ask-Elle, an IPT with an adept authoring tool framework. Finally, we discuss ITAP which is a data-driven IPT that can reliably provide adaptive feedback for 100% of learner submitted code.

For each we discuss contributions and limitations.

2.4.1 ChatGPT - Large Language Models for Education

In recent years, research centred around NLP[4] has trended in the form of "AI Language Models", and interfaces for them. Research on AI Language Models have advanced to the point where research has been applied to education and programming distinctly. Research in the field of NLP and responses may also pursue multi-modal inputs and outputs [73].

The most popular AI Language Model at present is ChatGPT, which is arguably considered a present state of the art "large language model" [74]. Research mentioning ChatGPT alone has more than octupled in publications between 2022 (with ~1280 results) and 2023 (with ~15,800 up to the date of this thesis) that can be searched on their respective dates on Google Scholar, or observed in discussions such as Ray et als. [74]. In this section, we briefly cover state of the art AI language models to distinguish their traits (strengths and weaknesses) from the focuses of our research. The example discussed is ChatGPT. We discuss ChatGPT in relation to programming education.

To function, ChatGPT requires training or train data. ChatGPT comes with an incredibly large data-set of train data. Through the OpenAI program and other ethical models [75], many data-sets are publicly available and growing [76, 77].

ChatGPT then facilitates conversation, taking natural language text (or even mathematics questions or programmed code) and responding with natural language text, numbers or code. The two most relevant areas to this research that ChatGPT has been applied to are "education", and "programming".

ChatGPT in Education

One notable discussion of ChatGPT is in education, where works such as Chaudhry's evaluate on circumventing cheating, or involving ChatGPT in education[78]. ChatGPT has demonstrated numerous contributions and ethical concerns when discussed within the context of education [76–78, 74].

For education, ChatGPT:

- Answers questions - ChatGPT produces answers to any queries it is trained to handle. There have been concerns of learners using ChatGPT for cheating on tests. In studies such as Chaudhry et al's. [78], ChatGPT's answers have been tested against pre-existing academic tests and entrance exams. It often would score approximately among what were the top 10% of learners. Clever use of ChatGPT can have it compose a question, a correct answer, and means to test an answer in a string to form a task.

[4]Research around having computers parse natural sentences and potentially respond with natural sentences. Knoll's work back in 2006 critically broke down and innovated on the aims and ideas in NLP [71]. A more recent example of NLP research is Mefteh's research [72]

- In practical use in education, ChatGPT can serve as an aide providing suggestions for:
  - learners studying or performing tasks,
  - tutors discussing and deciding "lesson plans, teaching strategies, and classroom management technique", [76]
  - adaptive navigation and suggestions for learning materials to cover - for the tutors or learners.

  These suggestions can then be observed, applied or ignored. [74] [76]

- A recurring stated challenge with ChatGPT is its lack of explainability - ability to convey the reasoning behind its choices. The reliability of the responses depends on the dataset, and the context ChatGPT accompanies. [74] This reliability additionally comes at the risk of losing the context of the original conversation in long conversations. However, although ChatGPT lacks explainability, it allows itself to be corrected by the user.

- ChatGPT's responses are also unable to be absolutely reliable. The precision and versatility of ChatGPT's responses and reliability is subject to training data, which requires consistent observation and data to bring ChatGPT closer to being reliable for any given context it is applied in. As ChatGPT lacks explainability, it makes it difficult to inquire on ChatGPT's reasoning. This makes it challenging to inquire if ChatGPT's response is correct, or to debug why ChatGPT reached an incorrect conclusion. Additionally, ChatGPT requires some responses to be phrased correctly to yield better results. [76].

- While ChatGPT can provide suggestions. However, suggestions in themselves can present problems. In general, providing too many hints at the wrong times can inhibit learning [30]; where Koedinger's work provides a discussion on finding the balance for the "assistance dilemna". For ChatGPT in specific, works also express concern that over-reliance on ChatGPT's suggestions in education may "inhibit critical thinking from over-reliance" [74, 76]. A concluding statement from a literature review in Kasceci's work [76] was: "while large language models can generate multiple-choice questions, produce text from bullet points, and scaffold learning, it is clear that they can only serve as assistive tools to human learners and educators and cannot replace the teacher".

ChatGPT in Programming

Along with education, ChatGPT supports answering queries for programming. Surameery's work [79] provides detailed paradigms on:

- potential uses of ChatGPT to support programming and development,
- problems ChatGPT may have in these areas,
- and concluding contexts where ChatGPT can be applied to benefit [79].

ChatGPT can be summarised to provide aide to programming in the following ways (summarised by Ray et al [74]):

- Code Generation - ChatGPT can be trained to generate working code solution chunks according to a given language, or alter code submitted to ChatGPT according to a request.
- Code Optimisation - ChatGPT can be trained to provide optimisation suggestions. This can form feedback on the quality and efficiency of the code if utilised in a programming environment.
- Debugging Assistance - ChatGPT can be trained to identify errors within code and suggest fixes when inquiring about an error. [79]
- Code Review - ChatGPT can be trained to summarise and provide legibility or "best practice" suggestions to code. This can form feedback on the layout and legibility of the code, whenever input into ChatGPT.

The extent ChatGPT would be able to do any of these, is subject to the trained, languages, code, contexts and corrections it was trained by.

ChatGPT Limitations

Despite these listed benefits in education and programming distinctly, they are rarely discussed in the context of "programming education".

ChatGPT expresses a lot of promise for use in programming education, but it is critical to know where, how, and what its limitations are. In this subsection, the current author broke down the limitations of ChatGPT and compares the strengths of IPT (precise, accurate, adaptive feedback). This is then used to discuss how ChatGPT can be used in support of education, and can be used in conjunction with IPT frameworks. [74]

The first limitation of ChatGPT in programming education is in sporadic precision. The output quality of the feedback is almost entirely based on trained data, topic of discussion, and how a query is expressed. This can sometimes output wrong information. Additionally, context can be lost over long conversations, or ChatGPT may fail to know the learning aims of a given task - potentially giving superficial contexts. Also, ChatGPT's responses are limited by what it can gleam from the training data. This overall leads to potentially imprecise or even incorrect responses. Lack of precision is a recurring stated limitation of ChatGPT being able to be applied alone, or relied on in educational contexts [74, 76]. However, articles also often state ChatGPT's uses as an aide in learning [74, 76, 79] - another tool to accompany learners, tutors or other tools.

A second limitation of ChatGPT for programming education is on its training requirements. As stated in the earlier section, without relevant training data in a given context, ChatGPT's aide can become broad, imprecise or unable to personalise responses for specified tasks - in every growing programming contexts. Additionally, ChatGPT needs to be monitored and further trained to improve how it is applied for educational contexts [74, 76]. Furthermore, this requires ChatGPT to be trained in using a given programming language. For custom programming languages, or domain specific code, acquiring public domain datasets becomes less likely. Ultimately, there's more of a requirement for one to train ChatGPT for any expertise in programming education they wish to apply it towards. Additionally there is no guaranteed results on its effects, and a requirement to monitor and further train it once deployed.

The last notable limitation of Chat GPT on programming education is due to its lack of explainability. The reasoning behind why a change should be made is critical for learners to observe future instances of a mistake and circumvent them. ChatGPT has challenges regarding "model explainability"[74] which refers to its ability to convey the reasoning behind why it concludes on an action. Any lack of explainability makes it difficult to convey the reasoning behind a suggestion. This impedes ChatGPT's ability to discuss the reasoning for its choices (although ChatGPT allows itself to be corrected).

Summary of ChatGPT

Overall, ChatGPT has several strengths and weaknesses. However NLP environments are not the focus of our research. The focus of our research is IPT with adaptive feedback frameworks and authoring tools for precise task creation, accurate feedback relating to the completion of tasks, and the facilitation of multiple programming languages. Without the need to train, monitor and further train data, IPT can theoretically provide easier task construction, even for remote custom languages. However, this would require facilitated authoring tools, and

versatility between programming languages (which are key targetted contributions of our research).

ChatGPT's applications are not mutually exclusive to IPT. It can theoretically be used alongside IPT. For instance it can work as a suggestion aide to accompany an IPTs programming environment, while the IPT provides precise and reliable hints, methods of checking task completion, and virtual environments that can be evaluated. It can also train using code of learners performing IPT tasks. Additionally, ChatGPT may be taught how to construct activities in the format of a given authoring tool. ChatGPT can also be run concurrently to software, like IPT.

Such tools are not mutually exclusive from adaptive feedback frameworks and IPT authoring tool frameworks - for providing syntax and semantic feedback. However, the focus of this research does not empirically cover using ChatGPT in conjunction with IPT.

2.4.2 Ask-Elle

Ask-Elle is an IPT for Haskell created by Gerdes et al. [18] They defined its field (IPT) as "ITS for the domain of functional programming". Ask-Elle supports class 3 definedness of tasks, and has a critical contribution in its versatile and easy to use authoring tool.

The authoring tool supports:

- The creation of tasks by simply entering one or more 'model' solution strategy codes.
- The ability to annotate over the solution strategies to define what text hint occurs at certain areas of code.
- The ability to specify compiled tests to be performed against learners solution strategy.

The aim of this section is to present existing IPT work for relative consideration of what can be a contribution, and what contributions can still be made in IPT. In this section, we first discuss how Ask-Elle works, and the tools used in order for it to function. Following this, we discuss the investigations performed on Ask-Elle by Gerdes et al. [18]. We then discuss the contributions and limitations on their approach.

Adaptive Feedback

Ask-Elle provides learners with its own programming environment, and tasks to complete. Each task can have one or more solution strategies. During a task, learners have the the following input options they can ask Ask-Elle:

- Check - to verify they are progressing to a correct solution strategy.

- "A single hint" - a next step to try, relative to what they have tried in code.
- "All possible choices on how to proceed" - the former, but towards multiple solution strategies Ask-Elle is aware of.
- "Worked-out solution" strategy- "presents a complete, step-wise, construction of a program" [18]

We note three phases to Ask-Elle's hint generation and solution strategy checking.

First, Ask-Elle uses the Helium compiler for program synthesis [80]. If it is unable to compile code, it reports the syntax error reported by Helium. In the final test case reported in [18], syntax errors were the majority returned feedback type by Ask-Elle. Syntax errors were returned 55.4% of the time, instead of semantic feedback, completion messages or no feedback.

If the code can compile, Haskell can identify how close learners are to a model solution strategy by 'comparing gaps' in the learners code relative to those solution strategy. Ask-Elle can also alternatively run the learners code to see if it succeeds in test cases with a model response if it fails.

For the test cases Ask-Elle uses Haskell's built in compiler, GHC to construct the program to be tested. To specify and perform the test, Ask-Elle uses the QuickCheck library [81]. The tests can also be used to add solutions that Ask-Elle did not consider.

Ask-Elle ultimately supports both model solution strategies *and* test cases.

With test cases, solution strategies are no longer static. This allows Ask-Elle to observe, assess and give feedback to run code, and is also used to provide the benefit of the tutor improving itself (were it to accept working cases). Though, doing so would potentially opens a security risk with the programs by accepting not tested (buggy) cases, or malicious solutions. However, the potential consequences of this risk are not observed.

With model solutions, Gerdes et al.list the following [18]:

- Any found equivalences[5] to solution strategies is guaranteed to be trustworthy. This is because model solution strategies are manually added. This is in contrast to test cases that can produce correct results, but be an incorrect algorithm, or lack quality.

  For instance, a bubblesort algorithm would produce the same results as a quicksort algorithm. However a task may want one to implement a specific on of the two. Matched model solution strategies do not have this ambiguity in correctness.

[5] an equivalences *after* program transformations to make irrelevant semantic differences be treated equally

- Imperfections can be recognised by model solution strategies and be reported. Ask-Elle comes with the additional features to format feedback for specific parts of solutions missing, and variants of solutions.
- Comparing model solution strategies to learner code means its possible to assume (or determine) which approach is being attempted.
- Testing by solution strategies is static. It does not require working code. The combinations are also preset. However, this is mutually exclusive from the benefit of test cases dynamically testing.

A downside of model solutions strategies is the fact they are finite. They require specification of all working instances. This is not as sustainable for larger programs, or more detailed or flexible programs. Gerdes et al.also believe this limits model solutions to introductory programming [18].

Our own research believes model solutions strategies can also be assessed in segments of code, rather than just entire code samples. This enables them to be used as aides in adding features, instead of limited to entire assessments of code. We facilitate this within our own IPT.

Authoring Tool

Ask-Elle facilitates the use of both model solutions, and test cases. A targeted contribution of Ask-Elle, was containing an authoring tool that easily facilitates the creation of both model solutions, and test cases. The use of model solutions strategies and test cases is not unique to Ask-Elle. However, a still outstanding contribution of this component was its ability to annotate feedback on code through pragmas, and annotate test cases and their responses.

The first step of Ask-Elle's authoring tool is simply to add Haskell code. From there, any number of annotations (none to indefinite) can be added. The 4 mentioned types of annotations in [18] are as follows:

- DESC - Allows a solution strategy to have an overarching description of what it's trying to do (or what the learner should aim to do for this solution strategy).
- FB - A location specific description. As it is parsed into the output AST of Helium, it can serve as a location specific description of feedback. The description used for a given point is the one deepest into the AST branches. This can be used to incrementally make hints more specific as learners progress through a learning activity.

- MUSTUSE - When checking for matching solution strategies, Ask-Elle facilitates some amount of variance. For functions, MUSTUSE specifies that the definition must match that of the solution strategy.
- ALT - Can be used to specify alternate lines or phrases for solution strategies, without needing to create entirely new solution strategies with those lines changed.

Of special note is the "FB" function which has a location corresponding to an AST after program synthesis. We hypothesise this feature can be done in a language independent authoring tool for any language supporting code comments or pragmas. Code comments would appear in their corresponding positions in an AST, where the comments could be used depending on where they are in the hierarchy, or where an "action" refers to. In the case of our research, we investigate how code comments are defined in ANTLR4 grammars. We do this not only to identify where learners use code comments, but also with the ideal of it being possible to specify custom hints or notes in a similar capacity to Ask-Elle, except in a language independent manner.

Ask-Elle can also specify properties and feedback for test cases. They're formatted by QuickCheck functions, and can ultimately specify:

- The program being checked,
- Where the output occurs,
- What values to check,
- How the values are checked,
- What feedback to return,
- What parameters to use in the phrasing of that feedback,
- Conditions to return that feedback...

...and more [81].

Ask-Elle Experiments

In this section, we go through 3 experiments conducted with Ask-Elle to then discuss their results. These are the 3 experiments mentioned in [18]. Each was ultimately a technical analysis and did not assess quality of learning. However, each involved learners in higher education.

The first of these experiments was a quantitative test performed in 2009. In it, Ask-Elle was used as an AAT, with a sample of 96 learner programs. The tool was tested to identify "correct" programs. All 32 "incorrect" or "imperfect" samples were not recognised, meaning Ask-Elle at this stage provided no false positives for solutions. This corresponds to one of the benefits of using model solution strategies.

The tool recognised 62 of the 72 correct solutions. However, to improve the recognition of solution strategies, the tutor would have to browse through the list of not recognised solutions, and add any observed correct ones.

The second of these experiments was a qualitative survey performed in 2011. It was run in a functional programming course with more than 200 learners, where Ask-Elle was an IPT.

The inspiration for adding properties and the tests was as a result of qualitative feedback saying that their solutions were not recognised by Ask-Elle.

Qualitative feedback also noted that response times were too slow, where Ask-Elle added a search mode for recognised steps. Ask-Elle still benefits from more solution strategies being defined, or more relative solutions to compare. It's possible this still limits the new approach to introductory size code examples.

The final experiment is a quantitative study performed in 2013. Ask-Elle was accessible online and would record a log of a learner's IP address, username and the "requested service" sent to Ask-Elle.

It retrieved 3466 interactions across 116 learners. This was also a technical analysis, where Gerdes et al.investigated how many requests Ask-Elle could respond to. The ideal was for the compilation, model solution strategy comparisons, or test cases to identify how every request should be responded to.

The rates were high where only 4.5% of interactions were discarded. However, they were not absolute even with the thoroughly improved programming activities. The activities could be retrospectively improved after the experiment. Additionally, 55.4% of requests were syntax errors.

Summary of Ask-Elle

Overall, Ask-Elle has the contribution of an easy to use authoring tool. Easy because it requires simply writing code, and annotating commands over it.

Syntax feedback and semantic feedback are mutually exclusive, where the majority of feedback is syntactic. This potentially hinders Ask-Elle manual descriptions from specifying how to steer learners away from syntax errors. However the syntax feedback produced by Helium and GHC should still help in Ask-Elle. While Ask-Elle's method of identifying the

nearest solution is expensive with larger changes, it is also consistent with the given feedback when helping learners iterate through gaps in a solution strategy. Aske-Elle is also limited to Haskell. They theorise their methods can be applied to other programming languages.

A limitation of Ask-Elle is it *requires* almost exhaustive numbers of the potential solution strategies and test cases to be able the return feedback consistently (yet still not absolutely).

Within our research's IPT, semantic feedback can be generated alongside any produced syntax feedback all of the time. This is done regardless of the number of solution strategies that have been defined. This is due to how program synthesis performed with ANTLR4's ALL algorithm. Unlike Ask-Elle, our IPT currently does not facilitate virtual machines or test cases.

2.4.3 Intelligent Teaching Assistant for Programming (ITAP)

ITAP is a "data-driven" IPT produced by Rivers [17]. It is an IPT that can provide syntax or semantic feedback for 100% of requests (regardless of the learner code and how a task is specified), and supports data-driven feedback. ITAP's contributions were in its ability to return correct feedback 100% of the time, and ability to improve the hints it provides quickly (without exhaustively covering solutions) by learners completing activities in ITAP.

ITAP consists of 5 major processes for hint generation for learners code:

- It generates an AST using a function built within Python3.
- The AST is converted into a reusable format when compared to other ASTs in a process Rivers calls "canonicalisation".
- Two canonicalised ASTs are compared to find a series of program transformations (actions) that can get from one AST, to the state of another. This is referred to as "Path Construction".
- The process of canonicalisation is then reversed in an AST, in a process Rivers names "reification".
- Finally, using the identified actions and the AST that had reification applied, ITAP conveys an action in a legible format, completing the process of "hint generation".

This process is explored in greater detail in Chapter 3, where the implementation of ITAP is compared to the implementation of our research's IPT.

ITAP was interfaced in 2 different ways. Earlier versions of ITAP was interfaced in the "Online Intergrated Development Environment". This version did not support data-driven feedback at the time. Unique to this version were 4 types of hints. These were:

- Next-Step hints - These adaptive and relative hints use the learners code, and compare to a relative solution. They return the "next step" and are the most common form of adaptive feedback in IPT. Early versions of ITAP had multiple levels of detail. Test results indicated that learners overall preferred more detail.
- Location hints - These hints simply refer to the locations of where the learner should change code. These were included as in face to face classrooms, there are instances where tutors would opt to point to where an error is, for learners to resolve.
- Example hints - These hints display solutions. These are included for cases where learners would look through learning material (or online) for how to implement a bit of code.
- Structure hints - These hints display the former, but referring to their token type. These are included as learners may want a hint from the start of an activity. These display an entire structure, but not necessarily how it's filled in.

In a quantitative evaluation with 28 recruited participants, Rivers found next-step hints were the most popular with 134 users across learners. For comparison, the second most of structure hints with 74 uses across learners. In this test and model of ITAP, a pattern was observed where learners with more experience of programming often stated they would prefer if hints displayed less detail. In those cases, it overtly told them the solution. In contrast, learners with less experience of programming would often state their preference of hints ideally having more information.

For such instances, location hints would benefit in instances where less information might be preferred, and example hints in instances where more information is needed. If a learner model could identify a learner's preference, they could be suggested automatically.

Learners generally stated a dislike of too much extraneous[6] information in hints. It appears to be critical to have as much information as desired, but ensure it is all intrinsic[7].

Learners also expressed dislike of instances where they were asked to "delete" and "replace" code they wrote. This may be because these prompts can request them to abandon what they have been attempting.

Of the 4 hint types, our IPT implemented next-step hints only. It would be theoretically plausible to implement the other hint types in a language independent manner.

- Location hints would simply use the location to a token.

[6]Where correct information not relevant to what one is considering
[7]Where all information is relevant to what one is considering

- Example hints would be to return a solution strategy.
- Structure hints could be constructed by writing out token types in order, with skipped or streamed formatting conserved. Structure hints could also be optionally formatted with a presentation of both token values and types.

Our IPT retrospectively did not consider implementing these hint options.

Another notable bit of qualitative feedback was that learners can sometimes know *what* to do, but not *how* to do it. Sometimes learners can also be shown an action to do, but be confounded as to *why* they should do it. In such instances, they claim they would search online or check through search sites. Two alternate ways IPT could facilitate resolving this are:

- The ability to specify the reasoning of implementing certain portions of code (or reiterate aims), like Ask-Elle's authoring tool features.
- Having a chat interface that can read and respond to inquiries (without context) with code suggestions like ChatGPT (instead of opening search engines).

Final version of ITAP

The later renowned version of ITAP also only had next-step hints. This was as a result of being implemented in a different interface. This change was to ensure it could deal with several users simultaneously using ITAP. This later version was built in the Open Learner Initiate [82]. It supported data-driven solutions (by automatically accepting completed tasks and using those for hints), and appeared to support test cases,

ITAP performed a number of technical analysis on its capabilities, and was verified to function as intended. ITAP performed a number of tests with real users. Their starting aim concerned "whether having access to hints during practice improved student learning". They found no correlation with this result initially, but revisited it in their final experiment. Their final experiment was with a real learner cohort across the academic year. Across the module, they had a pre test (before learning) mid term test, and post learning test. They also had special practice tests for the mid test and post learning test. In the final experiment, they wanted to evaluate if hints had an effect on the quality of learning. With an included sample size of 101 learners, they evaluated this by:

- Verifying if hints improvement on test scores. They concluded that no significant affect was observed. ITAP would not be the only means learners would have to improve their tests.

- Hints shortened time on practice activities. Here Rivers observed a notable 13.7% reduction in the time taken studying with the pre test practice activities. This result arguably has solid integrity, as there was no reduction in the subsequent test scores.

It is also worth nothing that as the academic year progressed, learners would go to older activities they had already solved to look at different code examples. Example hints can be used for such a context.

In terms of compilation speed, 97.5% of hints takes less than 5 second to calculate, given it is correct. Otherwise, 77.7% of the time, hints takes less than 5 seconds to calculate, and 9.4% of the time, hints takes less than 10 seconds to calculate.

Summary of ITAP

ITAP facilitates feedback 100% of the time, even with only one solution strategy specified. ITAP also supports multiple solution strategies, and the automatic addition of more solution strategies. However, ITAP feedback also can still only be either syntax or semantic. ITAP is also limited to Python2 and Python3. This is in contrast to our approach of a language independent IPT.

A notable portion of ITAP's research concerned trends with what motivates learners to continue to use an IPT through an academic year. Although a lot of observed trends are ambiguous, this is helpful knowledge on the presentation of IPT.

A strength of ITAP's experiments is that it was deployed in classes across an academic, which was observed to be rare in IPT empirical evaluations. Our own research did not create a version of our IPT to be deployed in an academic setting. However, our research still performs a technical analysis and user evaluation, with the latter similar to the recruitment survey of ITAP.

ITAP still uniquely supports data-driven additions of solution strategies for its reliable adaptive feedback framework, which is still an outstanding contribution.

It is idea for feedback to be responsive and near instant. Although our IPT uses the same path constuction actions as ITAP, our algorithms for path construction are unique. Our research also uses ANTLR4 for program synthesis. These two components may result in different hint calculation times to that of ITAP. Our research records our IPT's compile times.

2.5 Research Gaps

Through this section, we summarise each of the findings through each of the literature reviews.

2.5.1 Gaps in ubiquitous implementation of IPT

First, we have confirmed 3 specific areas in IPT are domain-specific.

1. The programming languages supported by individual IPT and their methods: Only a few secondary studies on ITS listed the languages supported by programming ITS. Our study only found automated assessment tools that supported more than one programming language, yet even in those instances, the method was both domain-specific and could not facilitate the easy addition of more programming languages.

2. Very little overlap between the implementation of programming languages in IPT has been observed. Hence authoring tools have been observed to be domain-specific. The 2 secondary studies critically discussing authoring tools evaluate that it would be ideal ("the future") for authoring tools to share methods and platforms they can work between.

3. The pedagogy surrounding IPT differs considerably: Our literature review hypothesised that pedagogy would not be discussed as frequently as needed by primary studies. It also hypothesised that they would not be compared as frequently as needed by secondary studies. Our literature review questions 2, 3a and 3b found neither of these was the case - pedagogy was frequently and critically discussed. However, the idea that the field has vastly differing implementation methods and surrounding pedagogy (outside of the unanimous inclusion of adaptive feedback) is supported by the observations of IPT in secondary studies.

Each of these is often stated as problems for time and cost to develop IPT around each language (as well as testing each IPT's pedagogical method) and authoring tools for each IPT.

By answering RA1, "Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?" we contribute to knowledge on a form of state-of-the-art adaptive feedback that can be implemented across programming languages, with the same built-in authoring tool method.

This is for the currently prevailing pedagogy in IPT - feedback. Our research focuses on adaptive feedback, but we have found further gaps in how other forms of pedagogy could be introduced into IPT. These include:

- Sensor-based technology to improve the student model of the learners.

- Generative learning (particularly self-explanation) to improve how learners memorise and try concepts.
- OLMs (and explain-ability of the models) to enable learners to inform their own student model.

2.5.2 Gaps in knowledge on the applicability of ANTLR4 for adaptive feedback

As no reference for categorising grammars was found, our research defined its own method of categorising the legibility of the tokens, were they to be used as feedback. This is to answer RA2b "What problems in grammar definitions can be listed/categorised that make feedback explicit? How can the phrasing of grammars affect the quality of feedback?".

It would be new knowledge to further research the phrasing of ANTLR4's syntax terms, how they can be used and categorised, as this is presently unexplored in literature.Using these, our research and further research could critically discuss how grammar can be phrased to improve feedback for IPT. We could also differentiate what impedes the benefits of feedback in the phrasing of the language grammar.

2.6 Chapter Summary

For this chapter, We first discussed the field of IPT, discussion in ITS that have yet to occur for IPT and recurring themes in IPT. OLMs and Sensors unobserved in IPT The domain/expert model was a common component used to define IPT, along with adaptive feedback. Cognitive Tutors were commonly defined to also have "pedagogical model" and a "learner/student". However, these components did not seem to occur in discussions with IPT. This supports our theory that a cognitive programming tutors have yet to be thoroughly explored.

Following this, we discussed the results of the second question on how programming languages affect IPT. Primary studies on IPT state the programming they use, however, we found that very few secondary studies (studies comparing ITS) specified the languages included, or focused on Computer Science or Programming. The few which stated programming languages found few multilingual automated tutoring tools were supported. Intellitutor eliminates language specific details through a process called "migration" (which descriptively resembles our proposed method), but was an AAT.

The next question our literature review performed checked for overlaps in research topics, or methods used across IPT. We found a lot of overlapping components depending on the field with adaptive feedback being a common trend and component for research in IPT.

The final question of our initial literature review discussed pedagogy in the field. Pedagogical discussion was found to be an overall prevalent topic of discussion. The pedagogy behind the methods used in the IPT themselves was almost unanimously included. The pedagogy behind how each IPT's research evaluate learning was very commonly included - reflecting on the quality of empirical research on the field. However, the individual definitions of what constitutes to learning was sporadic.

We also noted the following patterns from both primary and secondary research:

- The varying levels of class descriptiveness for programming language activities. The present highest in IPT requires activities with and that check "multiple limited solution strategies".

- The following feedback structure was presented in ITAP's feedback that use AST comparisons: [Location info] + [action verb 1] + [old value] + [action verb 2] + [new value] + [context] (Figure 30 in [17])

We then performed a small scale literature review on ANTLR4 uses in research, and follow with a discussion on the lack of research on ANTLR4 grammar applications. From the ANTLR4 literature review, a notable observation we made that is used later in this paper is the success of Wal's grammar sample working with 363 out of 370 grammars.

After this, we noted 3 related works: ChatGPT, Ask-Elle and ITAP. First, we quickly evaluated ChatGPT to provide contextual differences between NLP for programming education, and IPT. We noted ChatGPT's capabilities as an adept aide, providing programming suggestions for learners and tutors. However we also noted the training requirements and potential lack of precision for semantic feedback and syntax feedback. In contrast, we noted IPTs consistent precision and reliability for giving semantic feedback and syntax feedback. From this, we noted that NLP interfaces are not mutually exclusive to IPTs - they can work in conjunction. However, we emphasised that IPT are the focus of this research, and conjecture on how they can work in conjunction will not be empirically explored in this writing.

Next we evaluated Ask-Elle, an IPT for Haskell. It has an authoring tool that facilitates the specification of manually constructed hints by annotation model solutions. We note this serves additional motivation of evaluating ANTLR4 uses of code comments. This is because code comments could potentially be used for annotated solutions in all programming languages. We note frameworks supporting model solutions and frameworks supporting test cases each have their benefits. Ask-Elle supports both model solutions and test cases. Ask-Elle's authoring tool also supports the creation of both. For limitations, we note Ask-Elle's inability to guarantee feedback (even in cases of near exhausted solution strategies and test cases), and surplus of syntax feedback. We also note is is strictly for Haskell. Our

research has a contribution in supporting syntax *and* semantic feedback for all cases of learner submitted code, regardless of learning activity, for all programming languages.

Finally, we evaluated the IPT for Python 3 called ITAP. After specifying that ITAP's method is discussed in the next chapter, we evaluated ITAP's experiments. We list alternate methods that ITAP have for hints that are not next-step hints. We note a number of preferences learners were observed to have with hints, and when they were provided. We note the times recorded for ITAP to compile solutions in an earleir technical analysis.

Chapter 3

Research Methodology

A key aim of this research is to find in the context of IPT, "Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?".

Keuning et al. [25] noted "37.6%" of their surveyed works "use program transformations". A variant of this technique is referred to as "Migration" (as termed by Keuning et al. [25]) and has been used for a tutor supporting both pascal and C program [60]. "Migration is applied in INTELLITUTOR, which uses the abstract language AL for internal representation. Pascal and C programs are translated into AL to eliminate language-specific details. After that, the system performs some normalisations on the AL-code".

We proposed a method of implementing an IPT that refers to programming languages by abstraction. By creating and running such an IPT, we can investigate if it successfully returns feedback to any given learner code, for any given authoring tool created task for any given programming language.

The previous section outlines a number of classifications for IPT. This research will answer if the research aim is possible for:

- Class 3 type task definitions: With one arguable exception, this is the highest class definedness currently used in IPT. The exception is ITAP through multiple test users, and an interacting database of registered completed solutions. The method we chose facilitates the ability to add solutions (like ITAP), but for the experiment we will omit the process of dynamically adding solutions from a database - opting to manually define all available solutions instead.

  By creating a language-independent IPT with Class 3 definitions (and facilitate the ability to dynamically add solutions), we can prove the implementations of authoring

tool with definedness (Class 3 definitions) to be possible across all programming languages.

Task goals will be defined as pure text.

Tasks can have multiple of these goals set.

Languages will also contain the ability to list which token types to normalise - allowing inconsequential naming semantics to not be regarded as errors. Testing solutions can also check from any location within the code - if enabled. These two features prevent the need for exact matches of solutions, enabling Class 3 definedness; multiple solution strategies.

- IPT containing semantic Feedback: The majority of systems in Ngyen thinh-lee's study[1], 15/20 returned semantic feedback. This provides direct feedback on how to succeed in the given learning activity, rather than incidental errors.

  As "semantics", these need to be defined by context. This can be performed by "built in" authoring tools[1]. We can prove its viability for every possible programming language by including "intention based" - and ultimately, "code based" - feedback built-into a language-independent IPT .

  Code-based feedback can be performed in a language-independent IPT by:

  1. First having task goal code defined by an authoring tool.
  2. Second, having the task goal code compared to the current learner code ("problem" code/state).
  3. Third, context on the differences between the problem and goal state can be given, by comparing parsed representation of the code.

  These are the critical steps ITAP uses to provide code-based feedback[17]: "Canonicalization", "Path Construction" and "Hint Generation". However, a sizable portion of the process of "Reification" is missing due to its language-specific components. Reification's purpose helps ensure feedback is relative to the initial code. Reification also contains "layout" and "quality" feedback.

  Intention-based feedback can be performed by first having multiple task goals defined by the authoring tool, then having an algorithm to determine which one the learner is attempting to perform. Our research performs intention-based feedback by first executing the process of hint generation on each defined goal. During the process of

[1]Tools used to create tasks and features for ITS, without recompiling them[58]

"Path Construction", we assign 'weights' to each action. Weights are valued on an estimate of how many changes would need to be made for each action. The larger the weight, the larger the difference between the problem and goal code is estimated to be. Hence, the goal with the lower weight total (when compared to the problem) is assumed to be the solution the learner is targeting.

The viability of this method can be investigated on a technical level by attempting to perform different solutions for a task, and investigate if each targeted solution is correctly assumed.

- IPT containing syntax feedback: Even with successful semantic feedback, syntax feedback provides easy to obtain information on why the incidental state of a program would contain errors. However, to provide syntax feedback the IPT must be able to compile text for a given language, and return feedback on compiled text.

  Out of the 20 cases in Ngyen thinh-lee's study[1], 4 contained syntax feedback, where 2 of the 4 also contained Semantic feedback.

  Syntax feedback for any given language would be possible with, an IPT made in a way that refers to a compiler generator by abstraction. ANLTR4 runtime can be used to perform this. However, ANTLR4 can only return lexical and parsing errors. Tokenising and parsing may not involve the full compilation process for languages that assemble files, link files, contains pre-compilation changes, or require multiple compilations; unless language-specific code is created within the IPT. It is worth noting that ANTLR4 runtime supports the ability to "walk" through the compilation process to create accurate compilers, however this would require specified additions to such languages; by definition it would no longer be language-independent.

  Regardless, by default ANTLR4 runtime records potential syntax error corrections in parsed ASTs. ANTLR4 runtime can also have definitions of these syntax errors manually called. This is recorded in the generated ASTs,

We created an IPT to test with called Nue Tutor containing each of those definitions, that uses program synthesis.

3.1 Research and Evaluation Methodology

To perform a technical analysis for our language-independent IPT - Nue Tutor - is as follows.

ITAP's pre-canonicalisation, path construction and hint generation were proposed to be possible for IPT and programming languages outside the specific domain of ITAP. This,

method had yet to be proven for every possible programming language, which contribute to how many IPT works are language-specific (and hence domain-specific).

This section defines a language-independent implementation of pre-canonicalisation, path construction and hint generation, drawing parallels to how ITAP defines their implementation of canonicalisation, path construction and reification.

This is to provide a framework to test and evaluate:

1. if the syntax comparison of ITAP's state of the art implementation is *absolutely* possible for every programming language. Furthermore, if not, to list any occurring constraints and exceptions for the given method(s).

   This corresponds to the research aim 1a, which is finding - "Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?"

2. if there are additional language-specific settings[2] that could be defined alongside a tokeniser and parser for a language.

   This corresponds to the research question 1b, which is finding - "Are there instances of language-specific additions needed to return feedback to a language-independent IPT?"

This section will start by covering how our IPT (Nue Tutor) is created.
Second, the section will state how Nue Tutor evaluates the success of the method.
Nue Tutor was constructed in Unity for:

- easy building between Windows, Mac, Linux, mobile devices and other systems.
- the potential of integrating a framework for tutoring Games Programming, and executing code.
- and as a familiar preferred tool for the current author.

3.1.1 Program Synthesis through ANTLR4

Through this section, we make reference to ITAP's process surrounding canonicalisation. First there's the creation of ASTs, followed by anonymisation of the code to reduce the "great amounts of non-meaningful variation"[17]. ITAP also attempts "a suite of semantics-preserving program transformations" to "normalize the syntactic structure of the

[2] An option for each language that can exist alongside how each tokeniser and parser is added

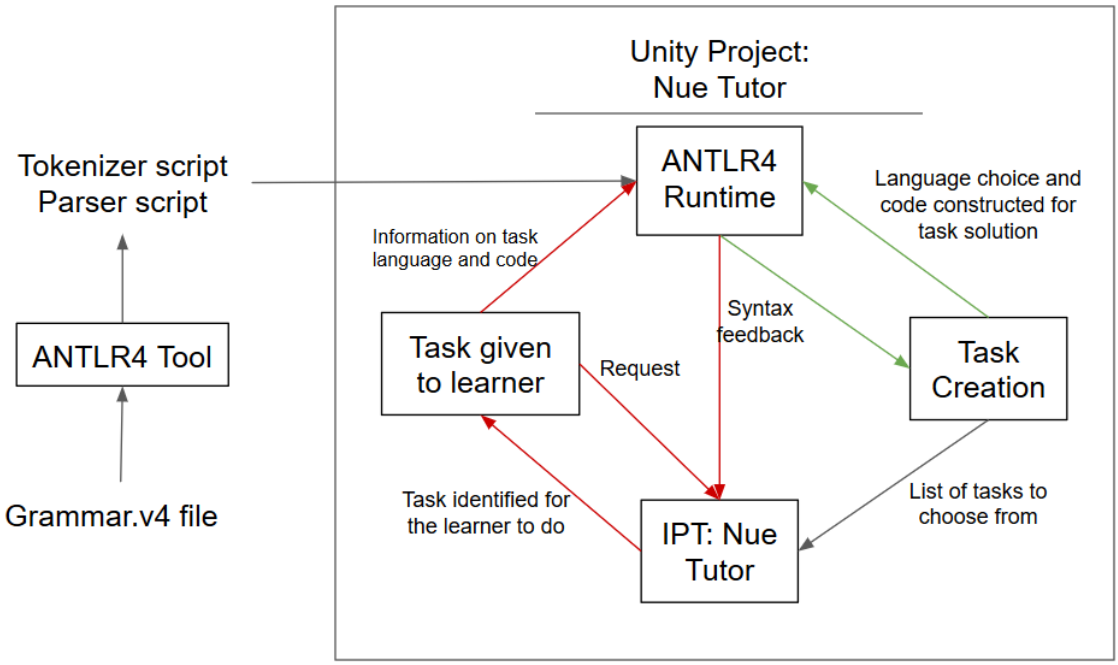

Fig. 3.1 An diagram of how the ANTLR4 tool and ANTLR4 runtime interact with Nue Tutor. "Task Creation" refers to Nue Tutor's Authoring tool. "Task given to learner" refers to Nut Tutor's interactive model

program (the code itself) while preserving semantic meaning (what the code does)" called 'normali[s]ing'[17].

ITAP also follows anonymisation, with "simplification" and "ordering". Nue Tutor does not include simplification or ordering. Their benefit however would be the same as anonymisation; improve how personalised the feedback is to what the learner is trying to do, and omit corrections that serve no benefit to the task. ITAP also explicitly categorises domain-specific features in the canonicalisation process.

For the process preceding canonicalisation (converting two sets of code to AST), Nue Tutor uses ANTLR4, created by Parr[45]. ANTLR4 was not created as a product of this research. However, this research makes use of ANTLR4 in the following ways for language-independent program synthesis. It is used for converting code to ASTs, that are then used by Nue Tutor.

There are two key components of ANTLR4 used outside and inside Nue Tutor respectively. The "ANTLR4 *tool*" and "ANTLR4 *runtime*". They are ultimately used by Nue Tutor to take any code (as text) as an input, and produce the same form of readable structure (AST) for Nue Tutor as an output. Using ANTLR4 for this process allows for all programming languages to produce this effect, once defined by an ANTLR4 grammar file and imported.

As all text languages can be defined, this theoretically should fulfil the process preceding canonicalisation for every programming language. Figure 3.1 contains a diagram of where Nue Tutor involves the ANTLR4 tool and ANTLR4 runtime.

My method for verifying the success of the feedback is covered in the next section.

Generating compilers and parsers with ANTLR4 tool

The "ANTLR *tool*" converts any file written for any ANTLR grammar convention into parsers and lexers of the grammar's language, in one of select languages.

The output of certain grammars could be specified to generate code that runs in a specific language, or domain which naturally prevent some uses being domain-independent.

ANTLR contains a GitHub repository of user contributed ANTLR grammars for various languages. To alleviate the number of domain-specific grammars that were constructed, ANTLR*4*[45] attempted to standardise submissions to be domain and output language-independent. As a result, the repository consists primarily of domain-independent ANTLR4 grammars. The grammars written in the programming languages of ANTLR4 are "g4" files. Nue Tutor runs in C#, and hence has the ANTLR4 tool output the compiler files as C# files. Hence, this standard helped provide the ability to sample many grammars on the repository, on Nue Tutor.

Importing languages is not "built in" to Nue Tutor, as they need to be recompiled upon importing a generated tokeniser or parser. However, the process only consists of:

1. Obtaining a grammar (a g4 file from the ANTLR4 repository, or created by a user)
2. Running the grammar through the ANTLR4 tool, generating 8 files.
3. Importing the files into Nue tutor (simply moving them into a folder Nue Tutor notices).
4. Linking the grammar's name to generic code.

Additionally, the process requires setting additional variables in given "prefab" variables for language-specific features such as what token types to anonymise. This latter action however *does not* require recompilation, so is arguably built in.

The generated files are also guaranteed to be compatible with the ANTLR4 runtime, given that they compile, and contain no domain-specific content in the source grammar definition (which has been standardised against).

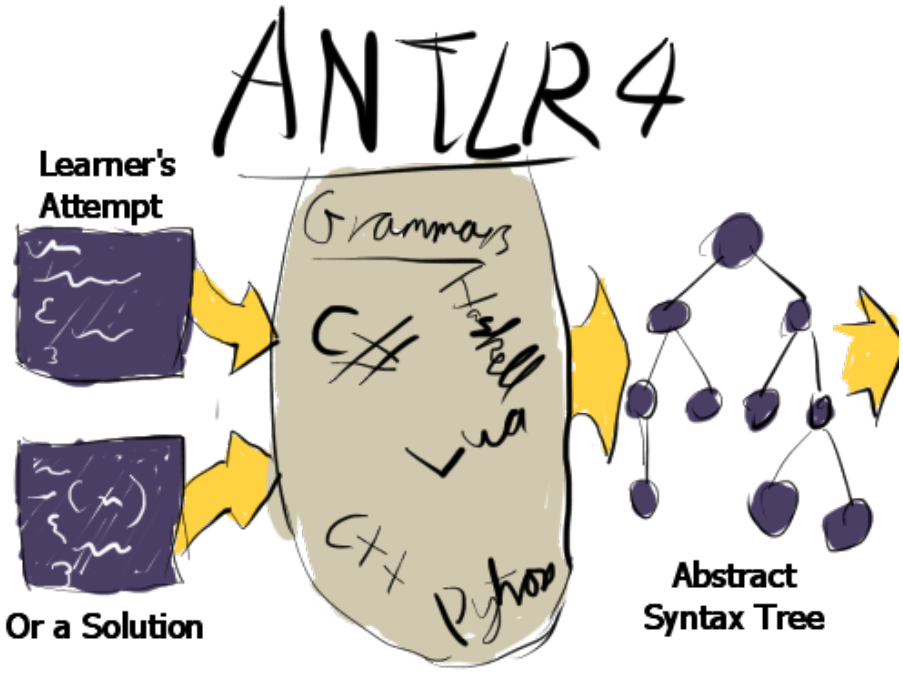

Fig. 3.2 An illustration of the usage of ANTLR4 relative to Nue Tutor. Code is parsed according to a given language and output as an AST, which is then converted and used by Nue Tutor

Program synthesis with ANTLR4 runtime

The "ANTLR4 *runtime*" defines the abstract base of the files generated. Nue Tutor uses the C# version of the ANTLR4 runtime for program synthesis.

The ANTLR4 runtime can be called by C# code to iterate through pure text to generate tokens by the convention of a given tokeniser. In a similar fashion, the runtime can be called to iterate through a series of tokens to generate an AST by the convention of a given parser. Calling a tokeniser and parser that correspond to the same given language, in sequence forms most of the process of Nue Tutor's program synthesis, to convert text to an AST of the given language.

However, for each tokeniser and parser pair, Nue Tutor needs 2 unique lines of code to explicitly refer to the class type of the tokeniser and parser. As these two lines needs to be compiled, this presents the core reason imported grammars by definition could not be considered "built in" to Nue Tutor.

The usage of the ANTLR4 tool and ANTLR4 runtime in this way forms the entirety of Nue Tutor's program synthesis - converting text code into ASTs in a language-independent way that is not built in to Nue Tutor. This part of the process is illustrated in Figure 3.2.

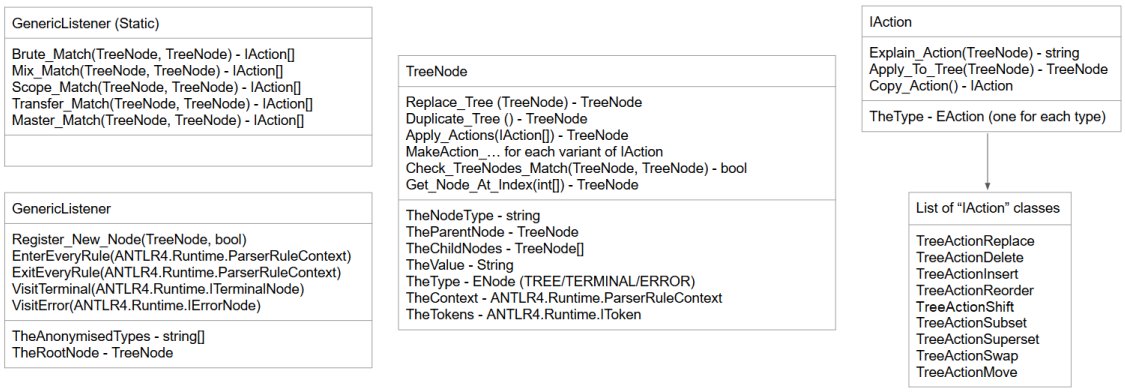

Fig. 3.3 The general structure of the class GenericListener and the classes GenericListener is responsible for. For each box: The top row refers to the class name. The middle row contains the class’ critical functions, where right of each function is their returned value type (if any). The bottom row lists a number of variable, with their type on the right. IAction is an interface with classes that inherit IAction listed below

Nue Tutor’s GenericListener

Nue Tutor ’observes’ the process of the ANTLR4 runtime generating an AST using a custom made "GenericListener" which is called alongside any language’s compilation. The GenericListener converts information as it’s being passed through into a differently formatted AST for Nue Tutor to use, modify and duplicate.

In the case of Nue Tutor, the following definitions apply:

- A Type - Is an enum description stating whether it is "Terminal", "Tree" (has children) or "Error" (Terminal node that describes either extra information or missing information).
- A Token Type - Is a string description on what token/phrase type it was identified as during tokenising and parsing.
- A Node - contains A Type, a Token Type, and a list of nodes it contains - referred to as children.
- A full AST - refers to a root Node. That is a Node with children Nodes, that is also not the child of any Node.

The classes involved during GenericListener’s observation of the programming synthesis are illustrated in the diagram; Figure 3.3. In the diagram, GenericListener shows the events triggered by the ANTLR4 runtime. Multiple "TreeNode"s form Nue Tutor’s modify-able

representation of an AST. Just before the ANTLR4 runtime is called for compiling code into a given language, GenericListener constructs a "TheRootNode" with no parent or children. From there, the first event triggered is used by GenericListener to populate the information in TheRootNode. In the previous section, we mentioned "Nue Tutor needs 2 unique lines of code to explicitly refer to the class type of the tokeniser and parser". These two lines are how Nue Tutor instigates the tokenising and parsing portion for a given language.

Subsequent event calls add TreeNodes, where each TreeNode must be a referenced "TheChildNode" of an existing TreeNode. Every child node has the node that they are a child of, set as their "TheParentNode". Following a series of references of "TheParentNodes" from any node will always lead to "TheRootNode" - with the exception being "TheRootNode" which has no TheParentNode.

GenericListener keeps track of which TreeNode should be modified next:

- "GenericListener.EnterEveryRule" will add a TreeNode to the currently tracked TreeNode, then make that node the currently tracked TreeNode. TreeNodes in this instance their "TheType" set as "TREE".
- "GenericListener.ExitEveryRule" will set the currently tracked TreeNode to its parent.
- "GenericListener.VisitTerminal" and "GenericListener.VisitError" will create a TreeNode where the "TheType" is "TERMINAL" or "ERROR" respectively. In these cases, the tracked TreeNode does not change.

In each case where a node is added:

- The text that forms a token is saved in the TreeNode variable "TheValue". If "TreeNode.TheValues" are read in sequence, it'll form a quotation of the code they're compilations of. However, it is critical to note that this excludes skipped or separately streamed tokens. For instance, if spaces are skipped, tokens will lack the spaces like so: "Forinstance,ifspacesareskipped,tokenswilllackthespaceslikeso"
- The named token type or named phrase type is recorded in the TreeNode variable "TheNodeType". This is performed by first pulling the type name from "ParserRuleContext.GetType().Name" (which returns a string). The way ANTLR4 tool names the respective token class types is in the structure of "[token name]+Context". To retrieve just the token name, we also subtract the last 7 letters of the string.
- A link to ANTLR4's representation of the information is also saved in the TreeNode variable "TheContext". Events pass this directly as "ParserRuleContext".

Ultimately, by Nue Tutor's instigation, the ANTLR4 runtime iterates through code, calling Nue Tutor's GenericListener as it does. Nue Tutors uses this to construct its own representation of the AST. The full tree can then be obtained from that GenericListener's TheRootNode. This forms the process preceding canonicalisation.

Anonymisation

Nue Tutor can then anonymise token types, which is a process used in ITAP's canonicalisation[17]. A number of token types can have what their text (their value) renamed, but have no significant affect on how code would run (given that each instance is renamed in the same way). Also, values can occur in token types that are synonymous with each other. In Nue Tutor, each imported language can have synonymous token types specified in a Unity prefab. This variable is an array of an array of strings. The strings refer to token types. The array containing the strings categorises each of those names as "synonymous". Token values that appear in one list appear in another. Finally, the array containing the array of strings allows for multiple independent groups of synonymous token types to anonymise.

If the synonymous tokens are specified, and Nue Tutor enables this feature, Nue Tutor anonymises learner code before it is checked with solutions. This call would precede path construction, where two sets of code are compared. For anonymisation, Nue Tutor iterates through both the learners code and a solution code, saving all values for the token types specified to anonymise. Were anonymisation to be enabled, Nue Tutor calls Path Construction with multiple combinations of those tokens matching the solution.

Path construction concludes with a weight value to compare what solution is more relative to a learners code. This value is also used to compare what combination of anonymised terms best matches the learner's code from a solution. Rather than test every anonymisation combination across every group synonym group, one synonym group has their possible combinations tested at a time (with the other synonym groups untested). From there, a final Path Contruction call is made, using the combination that had the least weights for each synonym group. Regardless, this process would still be comparatively expensive compared to the single call of Path Construction (per solution) without anonymisation.

In both the technical analysis and the user evaluation, this process was not enabled due to requiring knowledge of each language to specify synonymous token types. It was ultimately domain specific in nature. Additionally, this feature was not tested as thoroughly as the rest of Nue Tutor. Compile time would also greatly increase when performing path construction. The process would have to be called multiple times to calculate what combination of values would have the lowest weight.

It is also worth noting that anonymisation can not have a generic term with an incrementing suffix (like "global1" or "local2") across programming languages. This is because it is always possible to have grammars conflict with how they'd use token names or use of letters or symbols. However, it is viable to specify prefixes and incrementing suffixes for each language. Currently Nue Tutor's anonymises token values strictly to those either in the learner code or solution.

3.1.2 AST Comparison Path Construction

River's work, ITAP consisted of 7 "Actions" for modifying ASTs[17]. Rivers would identify a sequence of these actions that would change the current state of the code in AST representation, into the state of a goal in AST representation.

The 7 actions were:

- "Insert": Add a node.
- "Delete": Remove a node.
- "Replace": Change the information of a node.
- "Superset": Surround a node with information.
- "Subset": Remove information surrounding a node.
- "Move": Move a node.
- "Swap": Swap a pair of nodes.

It is possible to convert one AST to another using only a combination of "insert" and "delete" actions. The other 5 actions provide additional relative context to the code. Nue Tutor defines the same 7 actions in its framework. Their affect on a given AST node is illustrated in Figure 3.4. These actions take the form of "IAction"s in Nue Tutor's code, illustrated in Figure 3.3.

The order these actions are called is determined by distinct processes. Additionally whether they're preferred over other combinations of actions that reach some form of goal is determined by weighing each action.

Weighing a solution

One wants to refer to a solution that targets the goal most relative to what a learner intends to perform. One also would prefer to apply a series of actions relevant to what has been

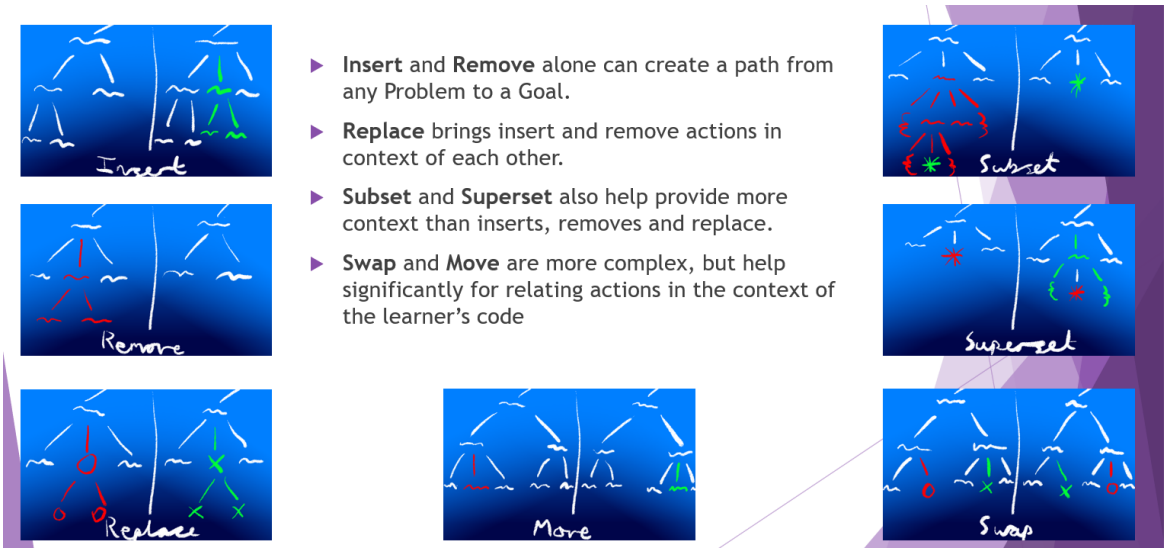

Fig. 3.4 An diagram of how the different actions affect the children nodes, relative to the parent node.

attempted. Both form critical steps for intention based feedback. Nue Tutor performs this by having multiple potential solutions, weighing them and choosing the solution with the least weight.

How Nue Tutor decides how to weigh each action is as follows:

- All actions increase weight with the amount of information modified.
- Weight tier 1 - Replace, Insert and Delete actions can not be considered relative if they have to modify large sets of information. The number of characters changed to the power of 4 is how these are weighted. Insert and Delete are then multiplied by 2. These are the "brute" actions.
- Weight tier 2 - Subset and Superset are contextual to what the learner has written, but still remove chunks of information, or add entirely new information. They are weighted by the number of characters surrounding the addition or removal, minus the persistent code, then put to the power of 3. As they are contextual, they have less of a power of than the brute actions. These are the "scope" actions.
- Weight tier 3 - Swap (and our variant "Reorder") and Move (and our variant "Shift") check the amount of distance between components in characters and puts the value to the power of 2. This makes larger distances have an increasingly larger consequence, but not as escalated as the brute actions or even scope actions. This is because they entirely reuse code the learner has already written.

Nue Tutor also uses these weightings due to occurrence of changes from the latter tiers being predicted as less likely. At present, Nue Tutor does not use a base weight value for the trees, unlike ITAP[17]. Nue Tutor only creates and compares the total weight of the actions.

Comparatively, ITAP had the following weightings for its actions[17]:

- "Change" (Replace): "take the maximum of the old and new value weights"
- "Add" (Insert): "weight of the new value"
- "Delete": weight of the old value"
- "Subset": weight of the new value - weight of the old value"
- "Superset": weight of the old value - weight of the new value"
- "Swap" (and Reorder): 2"
- "Move" (and Shift): 1"

River's aims were also "to emphasize how much of a change the student will need to make", and hence she *also* has larger weight differences on what we would categorise as the lower (larger) weight tiers. "Change", "Add" and "Delete" (our tier 1) use the full weight of the changes. "Subset" and "Superset" (our tier 2) use the weight difference. "Swap" and "Move" (our tier 3) use minimal points. Although the method for each weighting differs, the aims behind how this research and ITAP tier the weightings are identical.

3.1.3 Calculating a solution

Nue Tutor consisted of 4 key steps to calculate a solution on how a problem can reach a goal:

- Scope Match: This seeks out what can be reached purely by supersets and subsets.
- Mix Match: This identifies what can be modified within a node. This identifies relative swaps named "reorders" within a node. It also identifies relative moves named "shifts" within a node.
- Brute Match: This function identifies any remaining differences as insert, delete or replace actions. This function is called last, as it is guaranteed to find a solution, and involves the highest weight actions.
- Transfer Match: After a solution is found, this function then checks through for pairs of actions, to form "Move" and "Swap" calls.

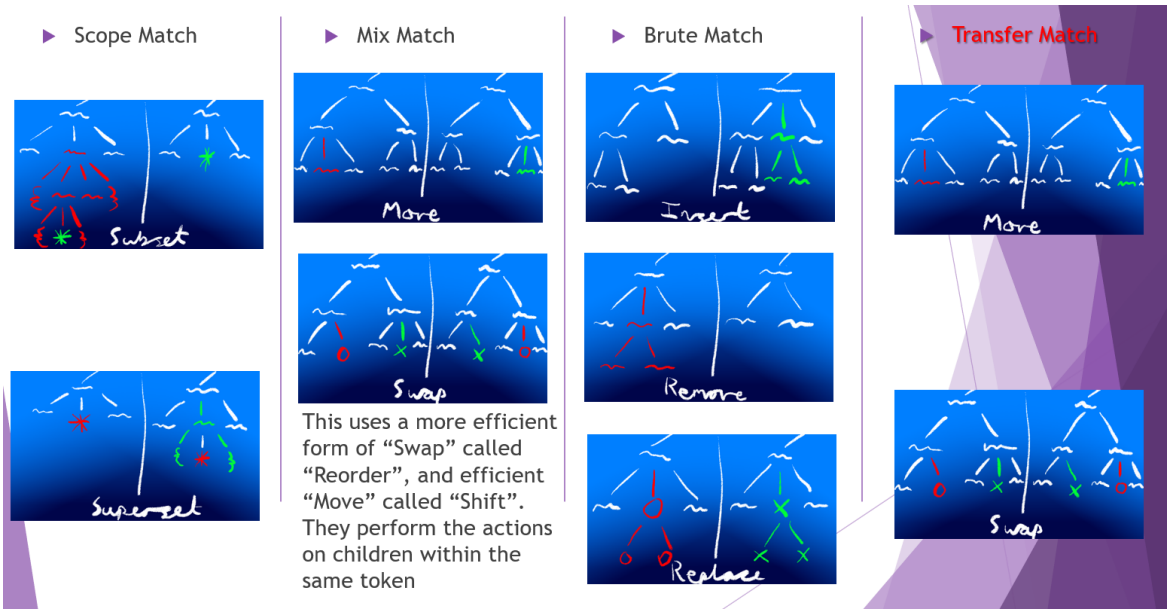

Fig. 3.5 An diagram of what actions the different algorithms in Nue Tutor find.

These functions are called in sequence and collectively output a list of actions that can get the problem to the goal with theoretically relative information.

At the end, the solution with the lowest weight is chosen. Figure 3.5 contains a diagram of the combinations of actions each function generally identifies.

When calculating for path construction, the changes are applied to the tree between actions.

Calculating Scope Match

The present version of Scope Match identifies which nodes to subset or superset. The process iterates (steps) through every node index shared between a problem tree and a goal tree. This function is recursive, passing a list of actions to be filled. We provide a flow chart of the process in Figure 3.6 and 3.7

For each node pair (between the problem and the goal) checked in a step, it first checks which tree has a greater maximum child node depth (depth being how many nodes away a given child node is from the root node). If the problem node has a lower depth than the goal node, it iterates (as a substep) through the children nodes in the problem node with a depth equal to or greater than the goal node's depth. For each of these nodes, an equivalence is checked and if one is found, this is identified to be a node to be subset. A subset action is added to the chronological list of actions, and the next step is taken.

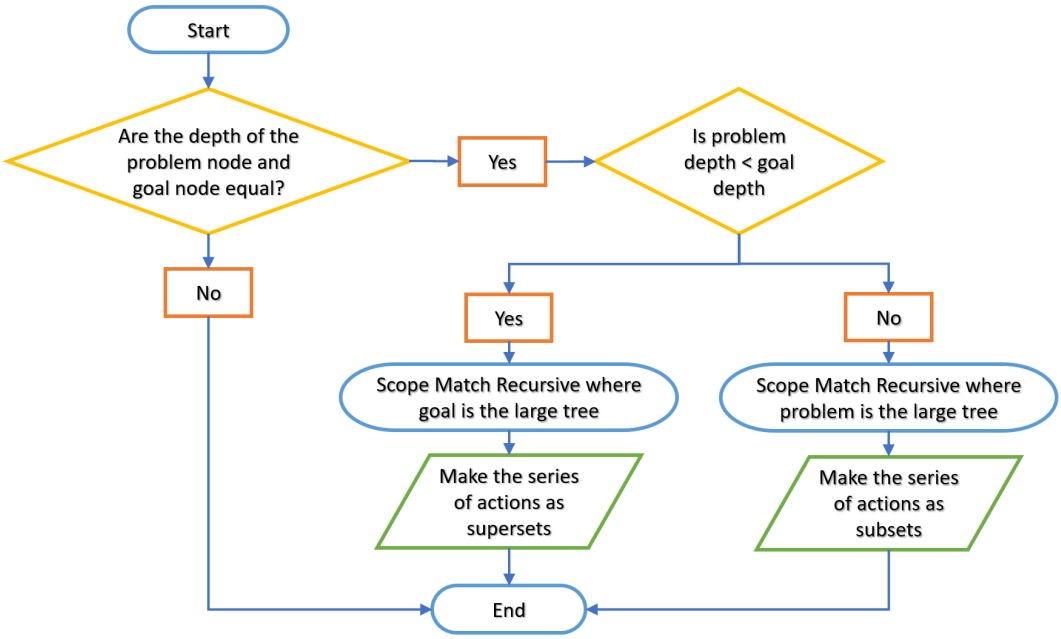

Fig. 3.6 A flowchart of when Scope Match calls the recursive function.

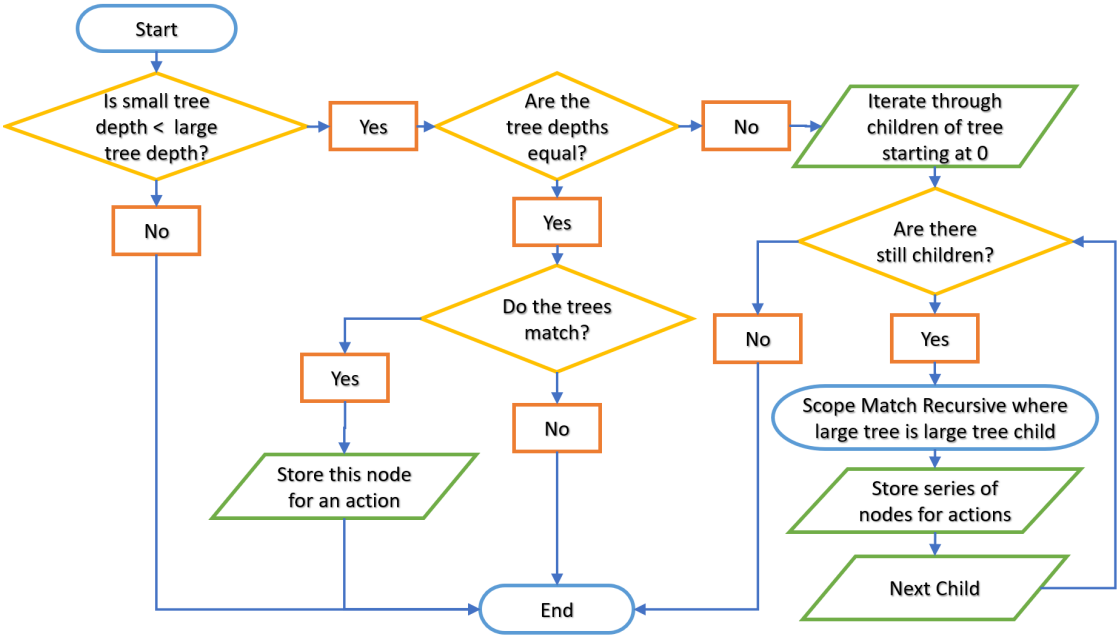

Fig. 3.7 A flowchart of how Nue Tutor iterates through an AST creates specified actions in the scope match function.

Problem State	Goal State	Action
((3),2,(4,5))	(3,7,((4),5))	subset 3
(3,2,(4,5))	(3,7,((4),5))	superset 4
(3,2,((4),5))	(3,7,((4),5))	

Table 3.1 Given these set of values, a scope match would identify the following changes. It does not ensure a full match by the end.

If the problem node has a depth higher than the goal node, it performs the check for the problem node in the goal node instead, to identify if a superset action should be added.

If for the step no superset or subset is identified (which also occurs if the depths are even), the process steps through the children node until either the number of the problem children nodes or goal children nodes are exceeded.

This identifies all instances of unmodified or moved code being superset or subset, that will not be moved or modified by further actions. Mix Match also checks for supersets and subsets that occur in the context of code being moved across a given child.

An example of how scope match would identify actions for code in a custom programming language is illustrated in Table 3.1.

Calculating Mix Match

The present version of Mix Match identifies supsets, supersets, inserts, deletes, replaces, shifts and reorders across a phrase/token. Just like with Scope Match, the process iterates (steps) through every node index shared between a problem tree and a goal tree. This function is recursive, passing a list of actions to be filled. A flow chart of the process is provided in Figure 3.8 and 3.9.

For each node pair (between the problem and the goal) checked in a step, it identifies an equivalent pairs of nodes between the problem and the goal (starting from the first) and marks them to be ignored. For any non matched child, it will then recursively perform this, and the following step for its children. After iterating through each child, it will then perform Mix Match between the problem and goal.

Mix Match will first check and list the matching indexes of pairs between the problem node's children, and the goal node's children (starting with the first index, where matched pairs will be ignored for further comparisons). Once this is performed, Mix match will check each unmatched pair for supersets and subsets via the same comparison mentioned in Scope Match. Pairs of supersets and subsets are noted, and the values for these pairs are also treated as regular pairs.

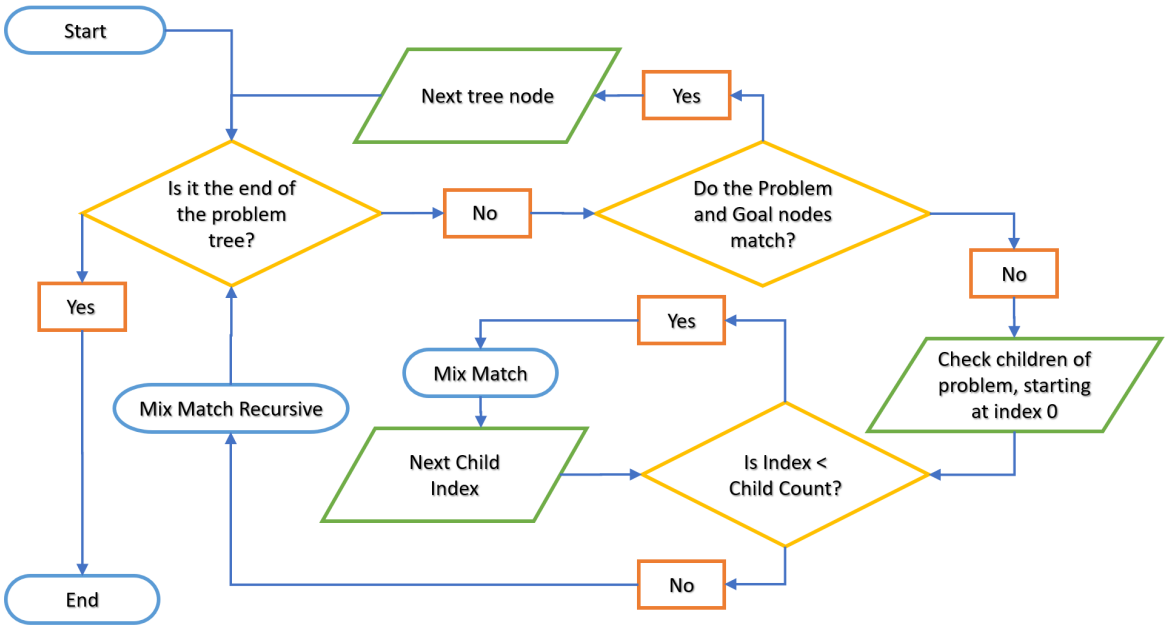

Fig. 3.8 A flowchart of how Nue Tutor iterates though an AST, to determines when it calls the Mix Match recursive function.

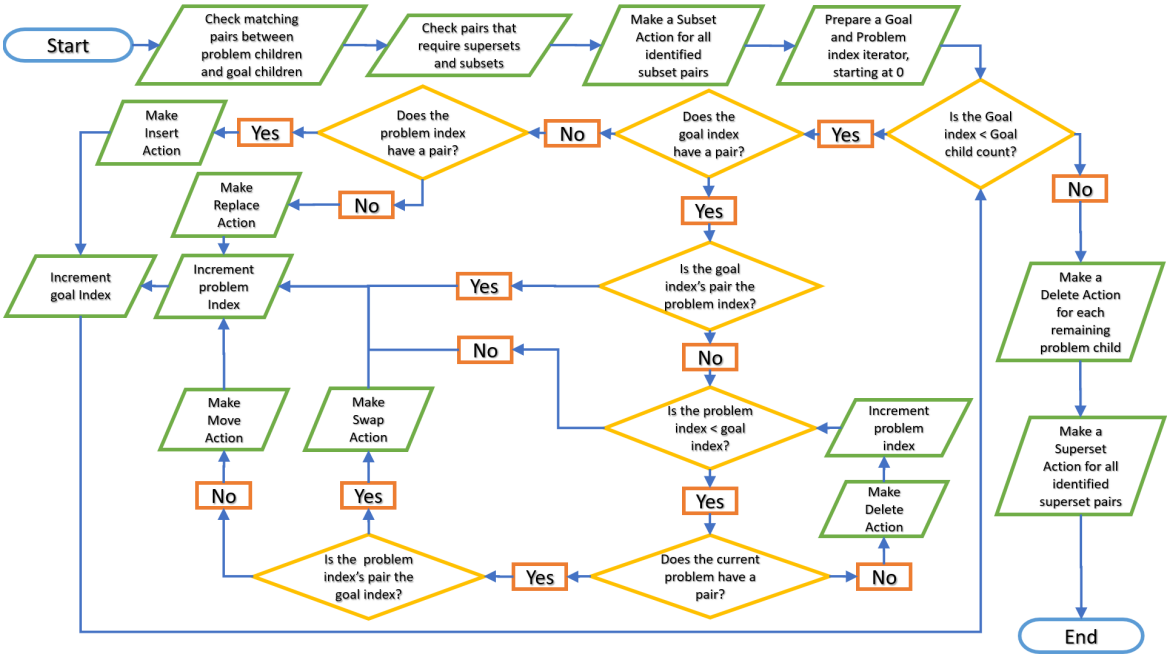

Fig. 3.9 A flowchart of how Mix Match's recursive function determines which actions to create as it iterates across a phrase.

Problem State	Goal State	Action
(((4),5),6,(3),2,7)	(3,(7),((4),5))	subset 3
(((4),5),6,3,2,7)	(3,(7),((4),5))	move 3
(3,((4),5),6,2,7)	(3,(7),((4),5))	move 7
(3,7,((4),5),6,2)	(3,(7),((4),5))	delete 6
(3,7,((4),5),2)	(3,(7),((4),5))	delete 2
(3,7,((4),5))	(3,(7),((4),5))	superset 7
(3,(7),((4),5))	(3,(7),((4),5))	

Table 3.2 Given these set of values, a mix match would identify the following changes.

For every subset pair, a subset action is added to the list according to the problem child's location.

From there, the aim is to create actions to have each child node from the problem, match where it would be in the goal. This is firstly done by having Mix Match store an index for the problem children to increment, and then iterate through the goal children. Here, first the goal child is checked if it has a problem pair. If it does not, it then checks if the current problem child has a pair. If not, a replace action is added to the list. If so, an insert action is added to the list.

If the goal child's index matches the current problem pair index, the next goal is checked. If the goal child has a pair, and the active problem is not the pair, then problem children are iterated through until the pair of the active goal is found. The pair of the first problem child is checked. Here, if the problem does not have a pair, and the number of problem children is greater than goal children, a delete action is added to the list. If the iterated through problem has a pair, and it's the active goal's pair, a swap action is added to the list and the next goal is checked. Otherwise, if the iterated through problem has a pair, but it's not the active goal's pair, a move action is added to the list and the next goal is checked.

If all problems or goals have been checked, any instance iniitally identified to be a superset then has a superset action added.

This performs all deletes, inserts, replaces, shifts and reorders. From there, the nodes recorded to be supersets are checked, and at the goal child's location, a superset action is added.

An example of how mix match would identify actions for code in a custom programming language is illustrated in Table 3.2.

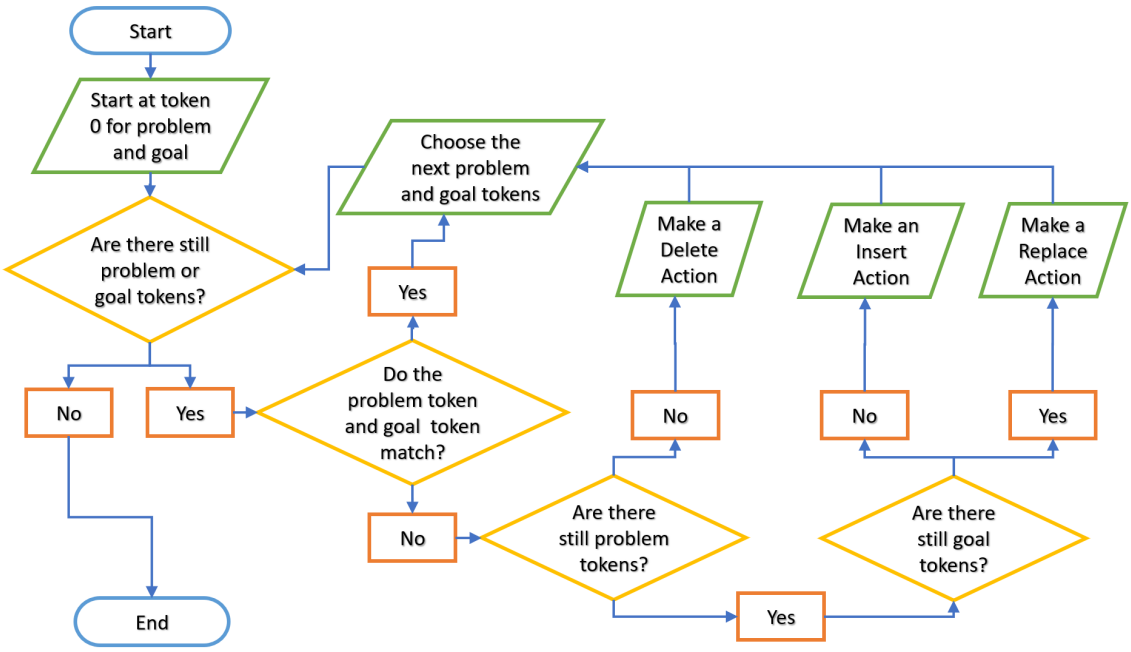

Fig. 3.10 A flowchart of how Brute Match iterates though an AST, to determines which actions to create.

Calculating Brute Match

This path construction action is called to guarantee any remaining differences between the goal and problem (after the current list of actions) are resolved. The process iterates (steps) through every node index shared between a problem tree and a goal tree. This function is recursive, passing a list of actions to be filled. Figure 3.10 contains a flowchart of the entire process.

For each step, an equivalence is checked. If there is none, a delete, replace or insert action is created respectively depending on whether the problem has more children, equal children, or less children than the goal.

This simple series of steps completes all path construction with inserts, replaces and deletes only.

An example of how brute match would identify actions for code in a custom programming language is illustrated in Table 3.3.

Calculation Transfer Match

Once Nue Tutor has obtained a complete list of actions that can successfully get the problem to the goal, the actions can be checked to find what can be combined.

Problem State	Goal State	Action
((4,5),6,(3),2,7)	(3,7,((4),5))	replace (4,5) with 3
(3,6,(3),2,7)	(3,7,((4),5))	replace 6 with 7
(3,7,(3),2,7)	(3,7,((4),5))	replace 3 with (4),5
(3,7,((4),5),2,7)	(3,7,((4),5))	delete 2
(3,7,((4),5),7)	(3,7,((4),5))	delete 7
(3,7,((4),5))	(3,7,((4),5))	

Table 3.3 Given these set of values, a brute match would identify the following changes.

Transfer Match iterates through the action list for pairs of actions that can be combined. The following pairs are checked:

- A Move can be made from an Insert that adds the same value that a Delete removes.
- A Swap can be made from a Replace with the opposite 'from' and 'to' values as another Replace
- A Swap can also be made from a Superset with the same raised and initial form as a Subsets initial and lowered form.

These functions perform all needed calculations to perform path construction by abstraction for all programming languages.

3.1.4 Hint Generation and Feedback

Following Canonicalisation and Path Construction, Nue Tutor internally has a list of steps that can get the present state of the learner's code to the state of the goal. The next step is determining a suitable way of presenting each action. The presentation and phrasing of feedback has been covered in the prior sections to be critical.

To prove the versatility of the method presented by ITAP for all programming languages (by having feedback in a comparable structure), Nue Tutor opts for ITAP's optimised 'full information' layout of: [Location info] + [action verb 1] + [old value] + [action verb 2] + [new value] + [context]

Nue Tutor interprets the terms as follows:

- Location - A co-ordinate in the code on where this hint/instruction is.
- Action - A verb stating what of the 7 actions to do.
- Value - A quote of either the problem (old) code or the goal (new) code.

- Context - the syntax type. The token or phrase the Value is contained in.

Nue Tutor provides location information, action term, value and context in different orders depending on the action. Each Action object has the ability to output feedback through the "IAction.Explain_Action" function shown in Figure 3.3. It uses information specified in the respective action. As it is an inherited function, how it presents information differs between IAction types. The token types (TheTokenType) for context, uses what the grammar calls a given token - which hence means grammar definitions can affect the legibility of feedback.

To reiterate, we investigate how it affects the legibility of feedback by answering RA2b, "What problems in grammar definitions can be listed/categorised to make feedback implicit? How can the phrasing of grammar affect the quality of feedback?". For the research in the paper, the resulting feedback of each hint action is presented in sequence all at once, as we evaluate the functionality of this method (rather than the affects of quality of learning).

The act of returning feedback hints completes the process of returning semantic feedback.

3.1.5 Authoring Tools

Nue Tutor contains a built in authoring tool for constructing learning activities. Given a programming language (that can be selected from a list), Nue Tutor tasks can contain multiple goals where each simply consist of text code written in the language. Nue Tutor allows for the root node to the specified, along with tests if the learner's code 'contains' the given goal and feedback on how adding the goal code could be improved.

Each task can add precondition (what skill strings are recommended from the learner approaching the task) and effects (what skill strings are added to the learner upon completing the task). This helps with recommending tasks to the learner and forms Nue Tutor's learner model. It is unused in the current build of Nue Tutor.

The built in authoring tool also allows for tasks to be given a 'starting state', which will load as the initial code state when a learner loads a learning activity. When comparing goal code to problem code, Nue Tutor will automatically normalise any tokens of a type listed in a programming language (if specified).

As the method only requires text (representing code for a given programming language), this authoring tool works for all given programming without needing to be reconfigured. Additionally (if normalisation is specified for a programming language), the problem and goal code only need to match: syntactically; and where the phrasing of tokens is consequential. This provides class 3 type specifications of tasks for all programming languages.

3.2 Evaluation Methodology - Technical Analysis

To reiterate, the core three RA stated by this research paper were:

- Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?
- Are there instances of language-specific additions needed to return feedback to a language-independent IPT?
- How frequently does the phrasing of grammars effect the phrasing of the context in feedback.

This section will cover the methodology on how this research considers these questions answered, and the methodology behind answering them.

3.2.1 Functional for every possible programming language

To approach investigating if the research methodology works for every possible programming language, the research aimed to test Nue Tutor against as many programming languages as feasible.

The ANTLR4 repository consists of more than 240 language grammars, and consisted of approximately 200 at the time of this research.

This research aimed to sample grammars of unfamiliar and familiar languages to the current author. It also aimed to sample multiple grammars of the same language to discuss if variations in defining the same language could affect the token definitions. Before running a technical analysis, we needed to import and verify the languages we would sample.

As manually sampling every language on the repository would be time consuming, every 20th language alphabetically was sampled. Languages that were familiar to the current author were then also included. If there were multiple grammars of a language, they would be sampled.

Each of the sampled languages would be downloaded and need to be tested for functionality. This would also require each language to have at least one learning activity made to run in Nue Tutor. This research avoided the need to study each language unfamiliar to the current author and craft activities for the language. This would be done by constructing a single learning activity for each language where each goal would be an example on the repository. These rules could be consistent across grammars, though this easily caused variance in the number of goals in each learning activity. This rule allowed for including cases with only one example, while still enabling testing how multiple solutions were handled by Nue Tutor.

No language had any of their language-specific variables set - anonymisation of variables was not set.

Each grammar had to be read through to identify what would constitute as a suitable 'root token type'[3] to be called by Nue Tutor. However, the root token type was not guaranteed to be the correct token type to call to start the compilation for each code. How code comments were also stored in each grammar was also investigated and recorded. We documented the number of root tokens, the root token ultimately used by Nue Tutor and whether Nue Tutor reported syntax errors with those tokens in their ASTs.

Whether each language was 'functional' in Nue Tutor was tested firstly by checking the composed trees in the authoring tool: for if they had unintentional errors, or a flat structure. It was also tested by copying the goal code, removing and changing a range of tokens, and submitting it as problem code - to see if the solution provided aimed for the corresponding goal. After this, each goal was then submitted as the problem. For each case, it was also checked if performing the initial sequence of actions would match a solution.

The results for each language were then recorded. Along with the former test, general notes were recorded on phrasing of the feedback for each language.

3.2.2 ANLTR4 grammar phrasing suitability for feedback

Our research defines context as the names used for ANTLR4 parsing terms. The classes for the tokens and phrases have consistent prefixes and suffixes according to what they are. Through the C# ANTLR4 runtime, the names of these can be called and cropped[4] by abstraction. If used for feedback, the phrasing of these terms would be judged by the user.

There is currently no standardisation of the parsing terms in ANTLR4 grammars. Hence, there is no definitive method of categorising the phrasing of feedback, or the impact of the patterns (what would make "feedback explicit"). Against each language, we categorise the notable differences of:

- how many parsing token names contain
  - partially or entirely abbreviated words
  - only full words
- how many parsing token names contain, 1, 2, 3 or more words

[3] A token that is not in any token's definition

[4] Remove the prefixes and suffixes to obtain the unique characters - which would form the token/phrase name

- how many unique words fall under different categories of "explicit", with each category being defined as:
  - Parsing Technical Terms - Terms that are understood in the context of the parser's usage
  - Abbrieviated Synonymous Terms - Terms used similarly to their recurrence in programming languages and computer science, but abbreviated
  - Relatable (Synoymous) Techincal Terms - Terms used similarly to their recurrence in programming languages and computer science
  - Language (Token string) Techinical Terms - Words or named symbols that appear in the language itself
  - Lit Tems (Abbreviated) - Words which can be understood by their literal meaning, but abbreviated.
  - Literal Terms (Unique) - Words which can be understood by their literal meaning alone.
  - Literal Terms (Descriptive/Adjective) - Words which can be understood by their literal meaning which are used to give context to other words in a parsing name.

  These are ordered from most domain-specific knowledge required, to least. Words that require relatable computer science knowledge are regarded as preferable to knowledge on a language's grammar definitions. This is because the user figuring out exactly what is meant by the specific phrasing of a language's grammar is superfluous to the language itself.

  Words that require language-specific knowledge (such as the names of string commands) are preferable to general relatable computer science terms. Until this is empirically explored, this research assumes users can gleam information on the context from observing terms used in the context that also occur in their code. Knowledge and naming used within the specific language the user is working on, is regarded as preferable to general knowledge.

  Words that are discern-able without technical knowledge are regarded as preferable to those that require knowledge on the specific language. This enables the context to be defined to the user if they are unfamiliar with specific terms. More specifically, descriptive words used in conjunction of other words.

- how each word is separated

  - lowercase
  - camelCase
  - under_score
  - under_score_end_
  - Single (Not Applicable as no token name contains more than 1 word)

  (All rule names start with a lowercase letter as this is syntactically required to define a "parsing rule")

These were the patterns expected by our paradigm, but there was a chance terms could be formatted in a way this paradigm did not account for. Hence this research searched through each grammar manually according to to the paradigm, meaning there's a potential 10% discrepancy of human error on the value counts. A notable retrospective option would be to use an ANTLR4 parser to read each parsing token type.

This research also constructed a paradigm for determining how the legibility of the languages are defined. This qualitative paradigm can then be referenced against the thoughts of users that read feedback[5].

The previous section stated what patterns are observed and listed for each grammar. These are used to then assign a keyword describing the legibility of the language. First, the number of unique words to form token type names are sorted to 3 groups.

- Implicit: The total percentage of words that are either "Parsing Technical Terms", "Abbrieviated Synonymous Terms" or "Lit Tem (Abbreviated)".
- Technical: The total percentage of words that are either "Relatable (Synoymous) Techincal Terms" or "Language (Token string) Techinical Terms".
- Literal: The total percentage of words that are either "Literal Terms (Descriptive/Adjective)" or "Literal Term (Unique)".

The terms are split between literal, technical and implicit based on how much knowledge of computing, the language and the grammar itself.

Secondly, the terms of a language are overall considered "descriptive" if the following 2 conditions are met.

1. More than 55% of the token type names contain more than 2 words.

[5] feedback produced by using the grammar's terms

2. The total percentage of "Language (Token string) Technical Terms" + "Literal Terms (Descriptive/Adjective)" is greater than the number of "Technical" words.

Using those two definitions, the paradigm for how languages can be sorted into legibility of use is as follows:

- Explicit: The grammar's names are descriptive, and more than 40% of the words are literal.
- Descriptive: Less than 40% of the words are literal, but the number of technical words and literal words each (counted separately) are more than the number of implicit words. The grammar's names are also descriptive.
- Literal: The grammar's names are less than descriptive, but more than 40% of the words are literal.
- Technical: Less than 40% of the words are literal, but the number of technical words and literal words each (counted separately) are more than the number of implicit words. However, the grammar's names are less than descriptive.
- Complicated: The majority (or equal) of words are Implicit, however the grammar's names are descriptive.
- Implicit: The highest (or equal) words are Implicit, and the grammar's names are not descriptive.

These are ordered from grammars with the highest legibility. The term rules are split between how self defining the names are for defining the context, and whether additional knowledge on the language or the grammar's phrasing is needed. Grammar with legibility considered "descriptive" are higher than those with legibility "literal" because technical terms are described and given context to in the former - also helping in tutoring the terms.

Checking this paradigm against the sampled languages should provide arguably the starting paradigm for sorting the legibility of grammar phrasing, that can then be evaluated and enhanced. This paradigm can then be checked against qualitative feedback on how well users can determine the context from feedback using each term.

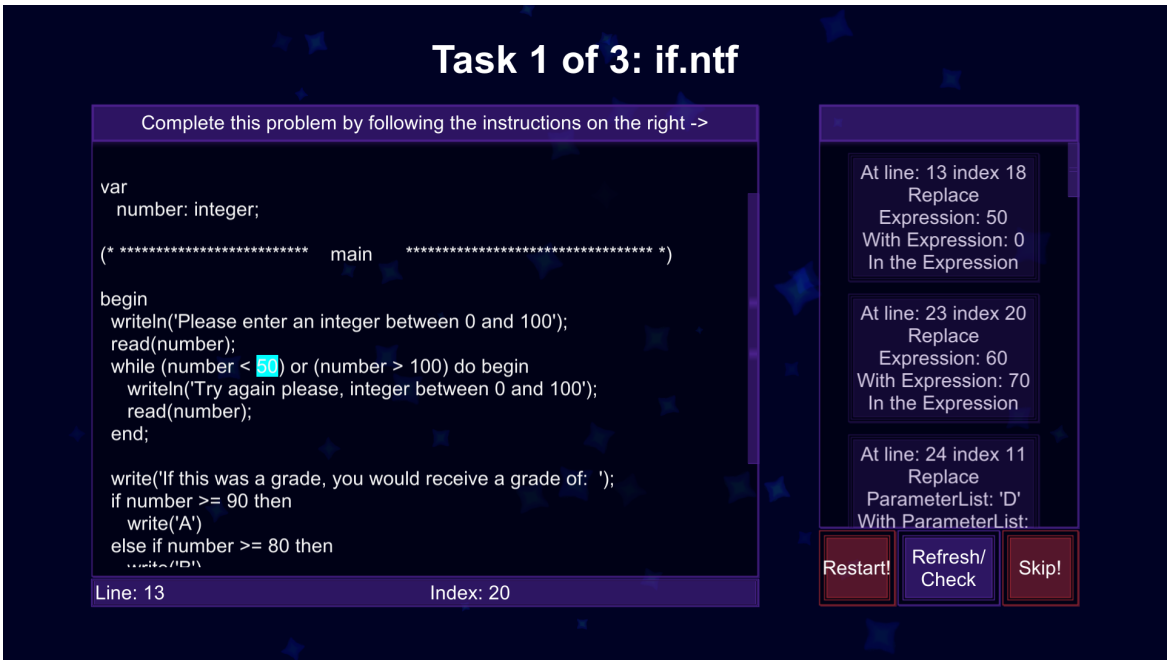

Fig. 3.11 And image of the special user evaluation build of Nue Tutor providing a mock learning activity.

3.3 User Evaluation

This research also sought users for qualitative feedback on the IPT's feedback, and to verify the validity of the legibility paradigm. The feedback can also be used to evaluate the helpfulness of the hints, relative to the proposed skill[6] of the user.

The key questions Nue Tutor investigates about itself are made to mirror that of River's[17] questions on feedback, excluding the questions on each learner's academic learning context. However, questions on each user's thoughts on their own skills on a language were collected. They were used to evaluate if it affected when a grammar's language legibility affected how easily hints could be read.

For the user evaluation, I created a unique build of Nue Tutor that facilitated the selection of programming languages, their learning activities and the questionnaires.

For each language, a number of examples are is pulled from the language's corresponding repository page. Using the authoring tool, examples on the repository of each language were then made into activities. Each activity had one goal and the initial state given to the learner being the goal state with a few modifications. This method allows the test to be easy to replicate for any language on the repository, for any sample (including languages we did

[6]how comfortable the learner declares they are with programming, and how comfortable the learner states they are with each specified language they used

Question	Possible Responses	Reason for inclusion
How would you rate your familiarity with each of the 3 languages?	1) I've never programmed before for ANY language 2) I never knew what's in it 3) I recognise general patterns in it from other languages 4) I've used, but just the basics 5) I've tried various features of the language 6) I've used the language adeptly	This metric can be used to compare how language legibility values are affected by user knowledge on the given language.

Table 3.4 The series of questions the user is asked after choosing a programming language

not sample). As the languages and example are also not refined for this test, this allows us to critically discuss problems with feedback that could occur naturally from programming language grammars.

To not overwhelm the participants with tests, the experiment gives the users 3 randomly selected languages from the sample of 14 to choose from. 3, 2 then 1 activities are respectively provided for the language. Before and after each language, a questionnaire is given. The provided questions and their purpose were as follows:

The pre-language question are listed in Table 3.4.

The post-language question are listed in Tables 3.5,3.6,3.7 and 3.8.

The interface presented to the users to answer the questions is shown in Figure 3.12

However, a qualitative survey was retrospectively not programmed into the user evaluation build of Nue Tutor and not inquired. This program was then distributed among friends and relatives.

The interface learners were given to complete the tasks is shown in Figure 3.11. Nue Tutor will also record the state of the code, when a "Check" is called. It will record each time a "Skip" "Check" or "Restart" are generated. Finally, it will record the full list of actions for each check called. This allowed the current author to document how many actions were generated as the user iterates through the task. For each language, the current author used the number of actions generated to:

- Estimate an overabundance of actions by the upper quartile of or the number of actions generated per check.

Question	Possible Responses	Reason for inclusion
You rated your familiarity with the language - [current language]] as "[previous rating]" What would you rate it now?"	1) I've never programmed before for ANY language 2) I never knew what's in it 3) I recognise general patterns in it from other languages 4) I've used, but just the basics 5) I've tried various features of the language 6) I've used the language adeptly	This metric can be used to compare how language legibility values are affected by user knowledge on the given language.

Table 3.5 The first series of questions the user is asked after completing a programming language

With instructions/hints, what helped you?		
Question	Possible Responses	Reason for inclusion
The line and index?	1) Never 2) Sometimes 3) Often 4) Overwelmingly	Inquires how useful the equivalent of the "Location" was in Nue Tutor for a given language.
The instruction verb (move, swap, insert, does not need to be in the...)	1) Never 2) Sometimes 3) Often 4) Overwelmingly	Inquires how useful the equivalent of the "Action verb" was in Nue Tutor for a given language.
The quote of your code	1) Never 2) Sometimes 3) Often 4) Overwelmingly	Inquires how useful the equivalent of the "Old/new value" was in Nue Tutor for a given language.
The context (the syntax type)	1) Never 2) Sometimes 3) Often 4) Overwelmingly	Inquires how useful the equivalent of the "Context" was in Nue Tutor for a given language. As this is tied to the phrasing of the grammars, this is also used for research aim 2b

Table 3.6 The second series of questions the user is asked after completing all the tasks in a programming language

Was the information in the instructions/hints legible or hard to read?		
Question	Possible Responses	Reason for inclusion
Were the words scrunched up (like with no spaces)	1) Always 2) Often 3) Rarely 4) Never	This value is obtained to separate formatting issues due to spaces, from other formatting issues.
Were the words abbreviated	1) Always 2) Often 3) Rarely 4) Never	This value is compared to the 'illegible word' percentage to evaluate how consistent user feedback is to this value.
Did the words elude to something you could not understand	1) Always 2) Often 3) Rarely 4) Never	This value is compared to the 'technical word' percentage to evaluate how consistent user feedback is to this value.
For hints in general, did you feel they needed to provide more information?	1) The opposite, they were too wordy 2) Whether there was too much or too little varied considerably 3) They provided enough information 4) Yes, they needed more detail	This is the first question on how users view any convoluted instructions. This response evaluates if the user felt if each individual instruction says more or less than what's needed.
For the number of instructions, do you feel they could often be phased in less steps?	1) The opposite, there were too few 2) Whether there was too much or too little varied considerably 3) They provided enough information 4) Yes, there didn't need to be that many	This is the second question on how users view any convoluted instructions. This response evaluates if the user felt that the number of instructions made following more complicated (by too few or too many joint together steps).

Table 3.7 The third series of questions the user is asked after completing all the tasks in a programming language

Overall did this language's instructions give a clear idea of,		
Question	Possible Responses	Reason for inclusion
What was what?	1) Never 2) Rarely 3) Often 4) Always	This is the first of two values for the user to state their overall feelings on the experience. This helps determine how much of an impact the other metrics have on the experience. In this instance, this inquires on whether the syntax terms and abbreviation caused confusion.
And/or a clear idea on what to do?	1) Never 2) Rarely 3) Often 4) Always	This is the second of two values for the user to state their overall feelings on the experience. This helps determine how much of an impact the other metrics have on the experience. In this instance, this inquires on whether the phrasing of the action verb causes confusion.

Table 3.8 The final series of questions the user is asked after completing all the tasks in a programming language

- Estimate the most common number of actions generated by the language by the medium of actions generated per check.
- The lower quartile of the number of actions generated per check is used as a control value, and a rough estimate on how far a user is from the goal before completing a task.

3.3.1 Evaluation Methodology

Each of these results will be checked to answer the research aims. Through this section we describe how we intend to use the results of these methods to answer the research aims.

RA1a - Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?"

Section 3.2.1 describes how the programming languages will be sampled. In the technical analysis, we recorded this as "Hints to Completion". If the success rate is unanimous for the feedback, and feedback returns in the given structure we could state this as true. Additionally, an unpolished build was given to users to evaluate. This build records the feedback returned with each press. We could further state this aim was true if feedback is unanimous returned, despite containing an unrefined sampled programming activities and many potential formations of code submitted by general users. We record how Nue Tutor responds to each "Check" call at the end of Section 3.3.

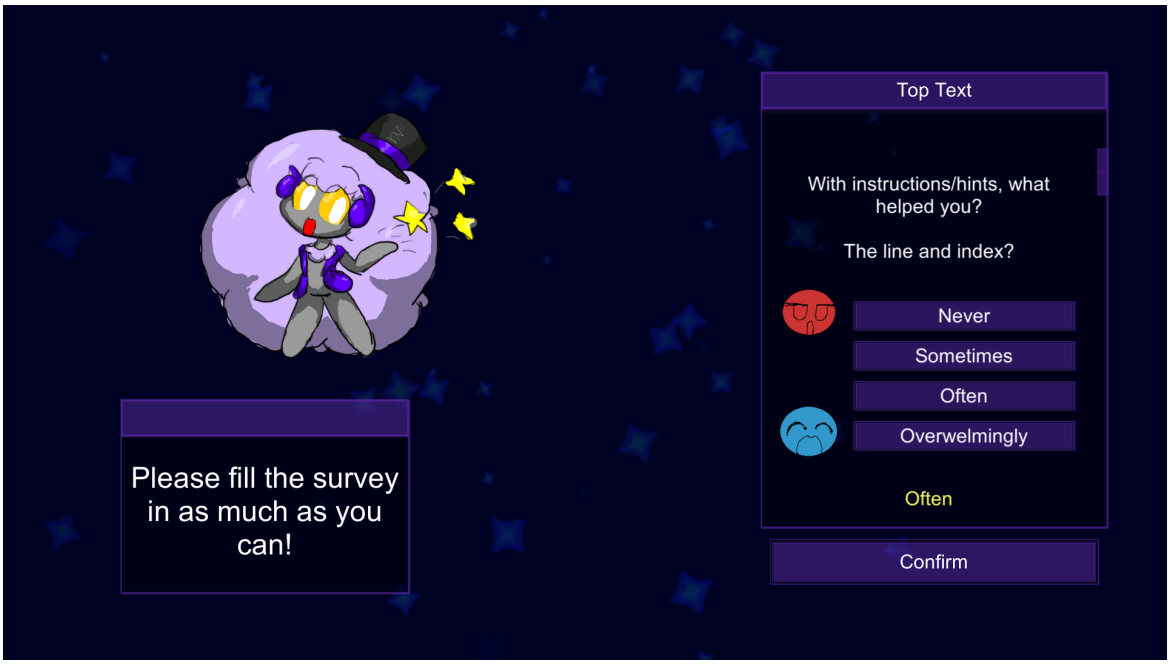

Fig. 3.12 And image of the special user evaluation build of Nue Tutor providing a post language questionnaire

This aim would additionally be successful if any exceptions can had their specific context listed. I hypothesised languages that would require multiple builds, or incorrect root token links to be an exception.

RA1b - Are there instances of language-specific additions needed to return feedback to a language-independent IPT?

We hypothesised potential reasons that there be language-specific additions for Nue Tutor. We record the root token type, and the type of syntax errors that occur for each grammar. If exceptions *do* occur, we test when they occur and additionally check these statistics.

If they occur due to needing multiple root token types, this implies a language-specific instance where multiple parses are needed for that grammar. If they present syntax errors, the problem is more ambiguous.

This result also promoted investigating what token types should be normalised in for each language, and how frequently. As this investigation was not conducted, an answer to the frequency of normalisation is not recorded. However, this method allows for concluding on how often the root tokens are not used, or the base language requires a more adept parsing process to return feedback on syntax and semantics.

If the majority do not have exceptions, we can still conclude 'that minimal to no language-specific additions are needed to function with the majority of grammars'. Were there more semantic variants of each grammar, we could also conclude 'regardless of how each language's grammar is written', but this requires more than one semantic variant to test.

RA2 - What impact can grammar phrasing have on the quality of learning?

We do not investigate quality of learning in this test, however we can investigate its affect on general legibility of the feedback. This Aim is split into two questions.

"RA2a - What are the root types that needs to be called for each lexer and parser? What types need to be called in sequence?" is answered alongside "Aim 1b". "RA2b - What problems in grammar definitions can be listed/categorised to make feedback implicit? How can the phrasing of grammars affect the quality of feedback?" is answered by the following:

Section 3.2.2 discusses how would assign each grammar to a paradigm based on their tokens. However, to check the affects of this phrasing, we compare this to how the users record their thoughts in the survey. I check if the "The context (the syntax type)" value and "Did the words elude to something you could not understand" value from Section 3.3 correlate with the "descriptiveness" ratings (word count, spaces, formatting, abbreviations and token legibility). If they do, it is conclusive on what affects the legibility. If the results are ambiguous, this question can not be conclusively answered, but the patterns worth further investigating can be drawn.

RA3a - What conditions in grammar definitions can be listed/categorised to affect the speed feedback can be returned? How does the compile speed change for different contexts?

Using the user evaluation, the ideal to answer this question would be to record the time between when each "Check" press is called, and when they are rendered for each language. This could also be checked against the code length (problem and goal) and the solution (number of presented actions). If the time is consistently below 100ms, this could be conflated with creating objects in Unity, or a simple tick for the frame and would be inconclusive. Such a result however would reflect positively on the compilation speed (being negligible, even for large code). If the time raises with the code size, we could attribute the raise to how code affects the speed. If the time raises with actions generated (or sporadically changes with grammars), we could attribute the raise to the intensity of the process of path construction.

However, this research failed to print the results with the data received from the user evaluation. As such, we can not provide a conclusion for aim 3a. There is however a list of languages that did not compile instantly which we will evaluate against the example sizes.

RA3b - What is the frequency of grammars including, streaming or excluding (skipping) code comments?

This critical statistic can be obtained directly from the sampled grammars' definition file. In case there were any ways comments were stored that were not accounted for by Semura's work[43], we list these. We present a final list on the percentage of each. The conclusion of these results is as they statistically appear. If skips and hidden channel streams are common, many base grammar files need to be modified. If channel streams are often used, we can conclude on the most often used term for code comments as a universal base to rename the comments channel to. If comments are commonly tokenised, this requires a more complicated solution to be addressed.

3.4 Chapter Summary

We developed a method that has the key components of ITAP[17], but uses program synthesis through ANTLR4 to work for any language we attach.

We developed the methodology to perform a technical analysis on this system, followed by a user evaluation to test the system, and provide feedback on the feedback.

At the end of this section, we describe how we would use each of these statistics to conclude on the initial research aims.

Chapter 4

Result Analysis

Through this section, we list the results we collected through our research, before relating them to the aims they were obtained from. This is followed by a discussion on each aim's answer, and a conclusion on their result.

4.1 Technical analysis

First are the results of the technical analysis on the sampled language grammars from the ANTLR4 repository, as well as these languages tested in Nue Tutor. When collecting the grammars, we also omitted a grammar if there were no given examples, or the language only supported single characters or phrases.

We sampled every 20th grammar on the repository alphabetically, after the 1st grammar listed. We additionally sampled languages the current author was familiar with (control samples), totalling to 23 grammars initially.

The control samples included were:

- abnf: At the time of constructing Nue Tutor, this was the first grammar on the repository. Nue Tutor was reconfigured to support normalising code. This grammar was the 2nd on the repository at the time of sampling grammars, and was re-included to test the affects of normalising on feedback.
- ANTLR: We use the examples listed which is consistent with the other samples. This is rather than the samples across the repository - which would suffice as examples. For instance, it would be possible for us to include one of the ANTLR4 grammars as an example for the ANTLR4 grammar.

- C++: This is a language the current author is familiar with, and has facilitated in an educational context.
- C#: This is a language the current author is familiar with, and has facilitated in an educational context. This is also the language the current author exports grammars into, and could use for tutoring Unity.
- Haskell: Ask-Elle was a tutor that used Haskell[18]. A solution for writing manual feedback through code comment, could be compared to Ask-Elle's method of manually writing feedback.
- json: This language was a case study (also noted in Parr's ANTLR4 reference book[45]) for ANTLR4's unique ability to construct grammars for languages that have multiple compile phases.
- Lua: This is a language the current author is familiar with.
- Prolog: The language used by the IPT commonly cited as the first well known IPT - Prolog Tutor.
- Python: This is a language the current author is familiar with. This language also enables a more direct comparison to functionality of the methods in Nue Tutor, to ITAPs. Python contains multiple semantic variants within each significant variant to compare.
- SQL: Language with multiple variants, and for handling databases. This would be a useful database language to discuss the effectiveness of feedback with normalisation in such a case. SQL also contains multiple significant variants.

antlr, json, python and sql contained significant variants[1] of their languages. This research opted for the most recent versions of the respective languages.

Bcpl (origbcpl), Python and SQL (MySQL and Oracle) were the only languages with semantic variants[2], however each variant had to be excluded as they failed to generate files in the ANTLR4 tool. Out of the two SQL variants, MySQL was tested. Hence, this research did not test for differences made by semantic variants.

- 2 were excluded for containing no examples on the GitHub repository.

[1] grammars for different versions of the same language (which could be argued to be a different language). For instance, ANTLR3 and ANTLR4, or C++ and C++11

[2] differently defined grammars for the same version of a language

Sampled Languages:				
abb	abnf	**antlr4**	bcpl	C++
creole	C#	erlang	**Haskell**	*HTTP*
json	**Kirikiri TJS2**	Lua	metric	pascal
prolog	properties	python	rpn	sql
stacktrace	toml	webidl		

Key:
2 no sample examples
Single char/line languages
G4 did not compile
Excluded at a later phase

Fig. 4.1 A table of the sampled languages, along with a coloured key for why each excluded language was excluded.

- 3 were excluded for consisting of languages with mutually exclusive or short phrases. (Grammars containing "Actions" and "in-grammar code")
- 3 were excluded as these grammars did not compile in the ANTLR4 tool.
- 1 was excluded as the resulting parser failed to create phrases. However this language was mistakenly left in the user evaluation build.

The following 15 languages from the ANTLR4 repository were included for testing: abb, abnf, bcpl, C++, creole, C#, erland, json, Lua, pascal, prolog, properties, python, toml and webidl. Figure 4.1 contains a table of all sampled languages. It highlights excluded languages in a colour corresponding to their reason for exclusion.

4.1.1 RA1a - Can abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?

The following results were produced to support answering whether "abstract syntax transformations function for every possible programming language to return feedback regardless of the submitted code (given a 'solution')?" in Figure 4.2.

Language	Examples	Rootless tokens	Used parser rule context	Tasks created	Node Type Coherency	Synthesis Legibility	Hints to completion
abb	1	module	module	No errors	Technical	Spaces Removed	Yes - Single
abnf	3	rulelist	rulelist	No errors	Technical	Spaces Removed	Yes - Each
bcpl	6	unless_command program	program	Required multiple parses Compiled and noted syntax errors	Explicit	Spaces Removed	No – Some goals required modification
C++	15	translationUnit	translationUnit	Compiled intentional syntax errors	Abbreviations Inexplicit	Spaces Removed	Yes - Confused priority
creole	7	document	document	No errors	Explicit	Spaces Removed	Yes - Confused priority
C#	1	compilation_unit	compilation_unit	No Errors	Underscores Explicit	Spaces Removed	Yes - Single
erlang	4	forms	forms	No Errors	Abbreviations Technical	Spaces Removed	Yes - Each
json	1	json	json	No Errors	Explicit	Retained	Yes - Single
Lua	2	chunk	chunk	No Errors	Literal	Spaces Removed	Yes - Confused Priority
pascal	16	program empty	program	Did not note errors for 1 intentionally faulty example	Technical	Spaces Removed	Confused priority overriding correct solution
prolog	4	propertiesFile	propertiesFile	No Errors	Underscores Technical	Unidentified	Yes - Single
properties	1	p_text	p_text	No Errors	Explicit	Retained	Yes - Each
python	26	single_input file_input eval_input	file_input	Compiled and noted syntax errors	Abbreviations Technical	Spaces Removed Tabs Retained	Yes - Each
sql	18	root (2702 lines - skipped)	root	Every phrase was a single token - Excluded	N/A	N/A	N/A
toml	4	document	document	No Errors	Explicit	Spaces Removed	Yes - Confused priority
webidl	1	webIDL otherOrComma	webIDL	No Errors	Explicit	Spaces Removed	Confused priority overriding correct solution

Fig. 4.2 Results for each language. The number of examples on the repository

The definition for each column is as follows:

- Example: The number of examples for the respective language, counted and downloaded from the ANTLR4 grammar repository.
- Rootless tokens: The number of token types in a phrase not within another phrase for the grammar's parsing rules.
- Used parser rule context: The chosen starting token type.
- Tasks created: Notes the type of syntax errors when copying code into the authoring tools, and if the errors are noted to be intentional as part of the design of the repository example.
- Node Type Coherency: How descriptive are the names of the tokens for use as a feedback context. This statistic preceded the descriptiveness paradigm created later
  - Literal - defines meaning of token types from literal descriptions, technical terms and a series of text terms used for tokenising. Multiple words form the rule names (more than 60% of the rule names). Words occur for multiple parsing rule names is a likely byproduct.

  - Explicit - rule type name refers to structure and context. Less than 30% contain literal descriptions or multiple words. Constructed from text terms used for tokenising, or technical literal definition.
  - Technical - knowledge of the internal language structure required, as terms are almost entirely technical literal definition, with single word rule names (at least 30%).
  - Implicit - knowledge on how this particular grammar is phrased required, as less than 30% of the rule names do not refer to even technical definitions, but technical names made strictly for the author of the grammar - with few recurring terms - or abbreviations of technical terms.

- Synthesis Legibility: How the formatting of the locations in feedback is affected by skipping or streaming tokens.

- Hints to completion: Whether hints always directed to a correct solution when followed. This was tested where each task had multiple goals. The term after the dash refers to how accurately it targeted a goal from a problem being that slightly modified goal.

  - Single - The created task for the language only had one goal (as the repository had one example code).
  - Each - When tested against each goal, the created task would correctly identify which goal was being targeted.
  - Confused Priority - When tested against each goal, a particular goal was prioritised over others, but the correct goal would be targeted when the code was close enough.
  - Confused Priority overriding correct solution - Some goals were heavily prioritised over others so even if a particular goal matched the problem, a certain goal would be prioritised instead.

Only one language had layouts that could cause it to fail to parse approaches to the goal; bcpl. This exception was due to the fact the language expected to be called for multiple passes for compilation, and all languages in Nue Tutor were configured to run once. The grammar was written in such a way that it would terminate before reaching the end of code.

Hence, languages that can conclude parsing before reaching the end of the code will not convert the entirety of the code into a full abstract syntax tree. This can then cause inaccuracies when comparing trees. An ideal fix would differ between context. For example,

languages can require a second round of parsing but read under a different grammar convention. This would be for cases that require pre-compilation definitions and macros that copy defined text. Languages can also require a different compiler at a different point (and hence be language-specific).

Otherwise, Nue Tutor successfully parses all states of learners code, all states of activity goals and a path between each. Additionally, of the 242 recorded checks called in the user evaluation in Nue Tutor, 239 (98.8%) successfully returned syntax and semantic feedback.

It is worth noting that abb had an exceptionally high action count automatically recorded from the user evaluation. This implies this language would not parse like the grammar intended. However the exceptionally high action count is surprisingly unique to it, in contrast to the other stated error languages.

Summary of RA1a

Firstly, not all languages we sampled were compiled. The potential issues for this could be due to using the ANTLR4 tool to build to C#, an error with the grammar files, or an oversight of this research. Grammar files with errors among each entry in the sample is unlikely, as Wal's[69] work successfully compiled 363 out of 370 (98.1%) grammars.

However, out of the grammars successfully compiled, the method almost unanimously worked against all programming languages - successfully returning semantic and syntax feedback across all languages regardless of errors in the problem code, or goal code. There were two observed exceptions to this. First was through usage of bcpl in Nue Tutor which required multiple compilation and would fail to return complete instructions as a result. The second was observed in the automated results for the user evaluation. In three instances, users clicked check, but neither more actions nor a completed task were returned. One instance of those was on a language known for requiring multiple passes of the compiler - JSON. I believe the other 2 instances may be due to an over-sighted error that we will have to explore in detail.

The success of the rate in which feedback is returned is supported in the user evaluation, where 239 of the 242 successfully returned feedback on how to get from the present learner's code state, to the goal code state.

For the accuracy of the weighting between languages, the wrong goals could be targeted in the 6 out of 10 instances of languages tested with multiple goals in the task. This could occur if the produced weights could be miscalculated to be negative, or 0. This figure was obtained in earlier versions of Nue Tutor before the formulas were further bug fixed and improved. The initial version could be concluded to not adequately weigh the actions. However, the present or future versions of Nue Tutor should confirm if this was incidental

to the prior version. Recording and comparing the exact values of the weights in a redone version of this experiment should help us confirm if these were weight calculation errors in earlier versions. If they were not errors, the recorded weights would help find what is incorrectly prioritised in Nue Tutor's method's calculation.

The following list summarises the adequacy of using ANTLR4 for program synthesis, Canonicalisation, Path Construction and hint generation to generate feedback for every possible programming language:

- It unanimously returns syntax feedback regardless of language. However, object compilation, file linking or other compilation specific traits would have to be specified.
- It unanimously returns semantic feedback regardless of language. As the same framework is consistent, it is language-independent.
- It unanimously performs syntax tree comparisons regardless of the state of the learner's code.
- It has an authoring tool that unanimously works with all programming languages.
- It requires identifying root tokens in a given grammar to use:
  - It may not parse all code (correctly or at all) if the root token is not correctly identified
  - Some grammars contained more than one root token
  - Some grammars may not start from the root token (root token may also be obsolete)
  - Some languages require multiple compilations, with a different starting token.
- Nue Tutor's older weightings worked successfully with 40% of the grammars, but we can not conclude on whether the current version of Nue Tutor's weightings will prioritise correctly for the other 60%.
- It may be possible to further improve semantic feedback through the skipped or streamed tokens.

4.1.2 RA1b - Are there instances of language-specific additions needed to return feedback to a language-independent IPT?

This research successfully performed syntax and semantic feedback, through program synthesis for all programming languages. While those were formerly identified to be domain-

specific techniques, through this section, we attempt to answer what can still be considered domain (or language) specific by answering the question, "Are there instances of language-specific additions needed to return feedback to a language-independent IPT?"

The implementation of Nue Tutor had 2 major language-specific constraints that had to be specified. First was the root token. Some languages also simply need multiple compilations (either through the same, or different root tokens).

Second was optionally specifying all token types that needed to be normalised. Our research functionally implemented normalisation, but this was not involved within either the technical analysis, or specified for each language for the user analysis.

Nue Tutor also had 2 forms of feedback that were not directly supported (where subsets and supersets could be considered as Reification and Layout Feedback for specific parts of languages). These were

- Quality feedback, as Nue Tutor does not observe the application running, or what would cause an application's running to be affected. This is because there is no runtime environment or virtual machine for executed code. Supporting a runtime environment is a helpful tool for constructivism (users being able to see the results of what code they have applied) for IPT.
- Layout feedback is also not supported. This was included in ITAPs[17], but Nue Tutor and any IPT using its method would require domain-specific additions - as layout changes between languages, and what would be preferable to layout itself is domain-specific.

Summary of RA1b

Overall, a list of language-specific attributes that would need unique frameworks to attach into a language-independent IPT:

- Root token - required for a language-independent IPT to function. Hence also functionally implemented into Nue Tutor. Adds a constraint for the user to either know the grammar, or to read and set the root token.
- Exceptions requiring multiple compilations - this has to be configured for programming languages that can not run by one compilation. We came across bcpl as an absolute required instance of this. This was the only example that would fail to function without this.
- Normalisation - functionally implemented into Nue Tutor, but needs an empirical test.

- Compilation or Virtual Machine - not implemented into Nue Tutor. ANTLR4 runtime (what Nue Tutor makes critical use of) supports the creation and running of compilers, or virtual machines.
- Quality feedback - currently supports no language-independent solution.
- Layout feedback - currently supports no language-independent solution.

4.1.3 RA2a - What are the root types that needs to be called for each lexer and parser? What types need to be called in sequence?

Through this section, we answer 'What are the root types that needs to be called for each lexer and parser? What types need to be called in sequence?'. For our test sample of languages, we did not research the grammars, but called what could be identified as a root token. Of the 16 languages tested, 12 contained 1 root token type. The terms used for root nodes were often ultimate: "program", "document", "compilation_unit", "json"(the name of the language), "file", "root". "program" and "document" occurred twice.

4 of the 16 languages contained multiple root tokens. Only 2 languages (which were 2 of the 4 containing multiple root tokens) commonly displayed errors in the generated abstract syntax tree.

Most languages would compile without error. bcpl had more than 1 root token type, and could fail. Python 3 had 3 possible root tokens, and while it was always successful in providing solutions, its generated trees often had coherent structures but would often contain error nodes in the tasks generated by the authoring tool. However, the rest seemingly only created error nodes when the code contained syntax errors - which is correct behaviour.

This was observed using Nue Tutor's authoring tool, which displays the generated tree for the activities, and highlights error nodes in red). Using the root token was, hence, majorly reliable but should be verified.

Figure 4.3 contains the observed results with the following definitions.

- g4 File Split - What files did the grammar consist of.
- Comment Action - How code comments are treated in this grammar.
- Rootless tokens: The number of token types in a phrase not within another phrase.
- Used parser rule context: The chosen starting token type.

Language	g4 File Split	Comment Action	Rootless tokens	Used parser rule context
abb	Lexer and Parser	Skipped	module	module
abnf	Single G4	channel(HIDDEN)	rulelist	rulelist
antlr	Mixed	channel(COMMENT)	N/A	N/A
bcpl	Single G4	channel(HIDDEN)	unless_command program	program
C++	Lexer and Parser	Skipped	translationUnit	translationUnit
creole	Single G4	N/A	document	document
C#	3 Part	channel(COMMENTS_CHANNEL)	compilation_unit	compilation_unit
erlang	Single G4	Skipped	forms	forms
Haskell	Lexer and Parser	Skipped	N/A	N/A
HTTP	Single G4	Tokenised+Parsed	N/A	N/A
json	Single G4	Skipped	json	json
Kirikiri TJS2	Lexer and Parser	channel(HIDDEN)	M/A	N/A
Lua	Single G4	channel(HIDDEN)	chunk	chunk
pascal	Single G4	Skipped	program empty	program
prolog	Single G4	Token	propertiesFile	propertiesFile
properties	Single G4	channel(HIDDEN)	p_text	p_text
python	Mixed	Skipped	single_input file_input eval_input	file_input
sql	Lexer and Parser	channel(MYSQLCOMMENT) channel(HIDDEN)	root (2702 lines - skipped)	root
toml	Single G4	Tokenised + Parsed	document	document
webidl	Single G4	channel(HIDDEN)	webIDL otherOrComma	webIDL

Fig. 4.3 A table of the sampled languages, along with a coloured key for why each excluded language was excluded

Summary of RA2a

Root tokens to operate a grammar can more often than not be a literal root token type (a token type that is not included in any other token type). This is particularly reliable way of finding the root token as most grammars contain only one root token with a definitely ultimate name. However, finding root tokens in this way is not absolute and may require deconstructing or a more thorough reading through the grammar. Additionally, some grammars may need subsequent re-compilations, and knowledge of that requires thorough knowledge on the compilation process of that language.

Ultimately, the amount of domain-specific knowledge needed to find the root token can often be circumvented by finding and using a single root token. Most programming languages have a single root token, but to figure out the root taken requires documentation, knowing the grammar, or reading through each token to find one that does not reoccur.

4.2 User Evaluation

From this section on-wards, the User Evaluation was involved in our investigations.

Legibility Rank	Occurences	Languages
Explicit	1	C#
Descriptive	3	C++, pascal, webidl
Literal	2	properties, toml
Technical	6	abb, bcpl, creole, json, prolog, python
Complex	1	Lua
Implicit	2	abnf, erlang

Table 4.1 The number of grammars that had each rank of legibility according to the rubric mentioned in Section 3.2.2

4.2.1 RA2b - What problems in grammar definitions can be listed/categorised to make feedback implicit? How can the phrasing of grammars affect the quality of feedback?

Through this section, we answer the research aim 'What problems in grammar definitions can be listed/categorised to make feedback implicit? How can the phrasing of grammars affect the quality of feedback?', with the information gathered from the technical analysis, and user evaluation.

Technical Analysis Evaluated Legibility

15 of the 21 grammars we collected were included in final usage.

The numbers of each legibility rank (and the languages with them) corresponded to the method of obtaining the legibility of each language in Section 3.2.2, and were as follows in Table 4.1;

10 of the 15 languages had less than 30% of their token names contain abbreviations.

This overall produces a large range of language legibility, with the median being technical (with a skew towards literal or descriptive).

An exception to the activities provided for each language was for C#. The repository did not provide any examples for the given version of C#. Rather than omit C#, it was included as an exception as the only "Explicit" grammar. The environment for writing Nue Tutor was full of C# files. As such, the files used in place of the examples were smaller C# files in Nue Tutor. The activities were made by copying the code as a goal in the authoring tool, and making a tweaked copy the starting state of the code - just like all the other languages did for the examples.

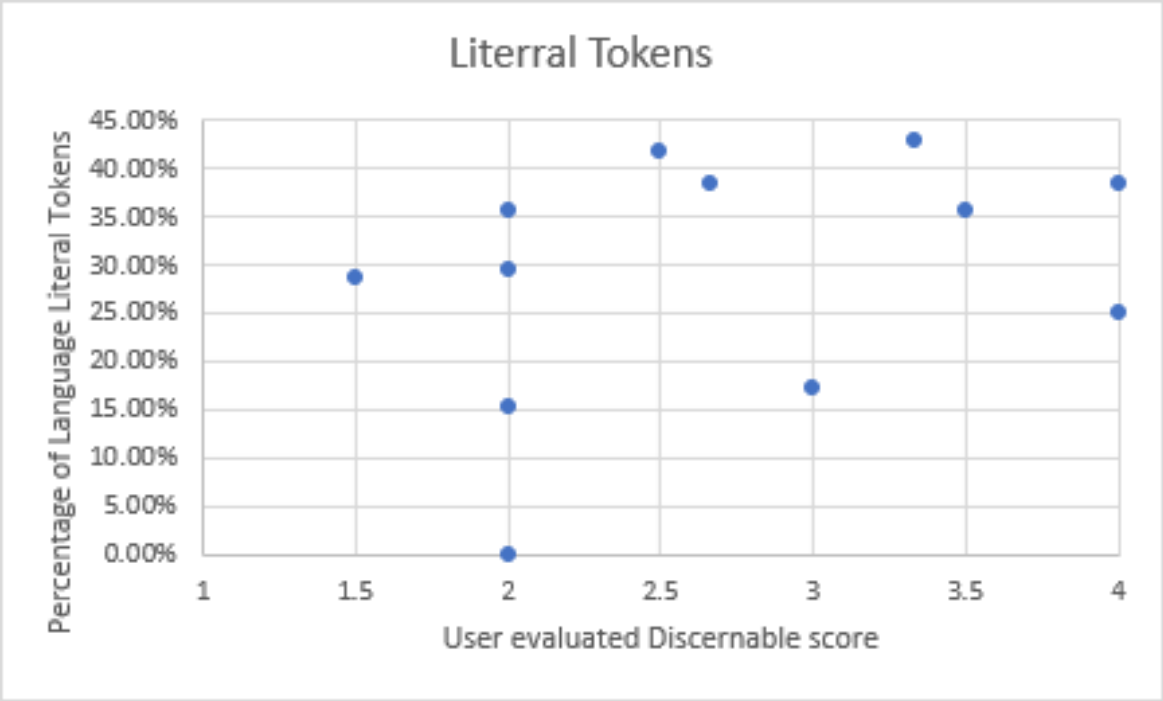

Fig. 4.4 A graph of the scores between what users rated as "discernable" and the descriptiveness score.

User Evaluated Legibility

Of the 10 participants that returned the result files, 2 participant's results were excluded for selecting Skip as the first action on every task. As no changes to the code were tried (where they could see how the feedback would respond), it would be unlikely the feedback would have been analysed, which could skew results. In contrast, participants who clicked check only once before skipping were included. 1 other participant's results more was excluded due to not completing each survey beyond 2 questions.

Figure 4.4 contains a scatter graph of the user's ratings for how "discernable" hints in a language were, against the number of literal tokens type names in a language.

Completion Success rates

Through this section, we discuss the completion rate of tasks for a given language, in relation to other statistics.

The first statistic I will note in this section are the familiarity scores. 4 of the 7 participants recorded that they had never programmed in any programming languages before, reflected by them stating an average familiarity score of 1.5 or less (meaning they specified they had not used *any* programming languages for at least 1 language). 8 tasks were successfully done

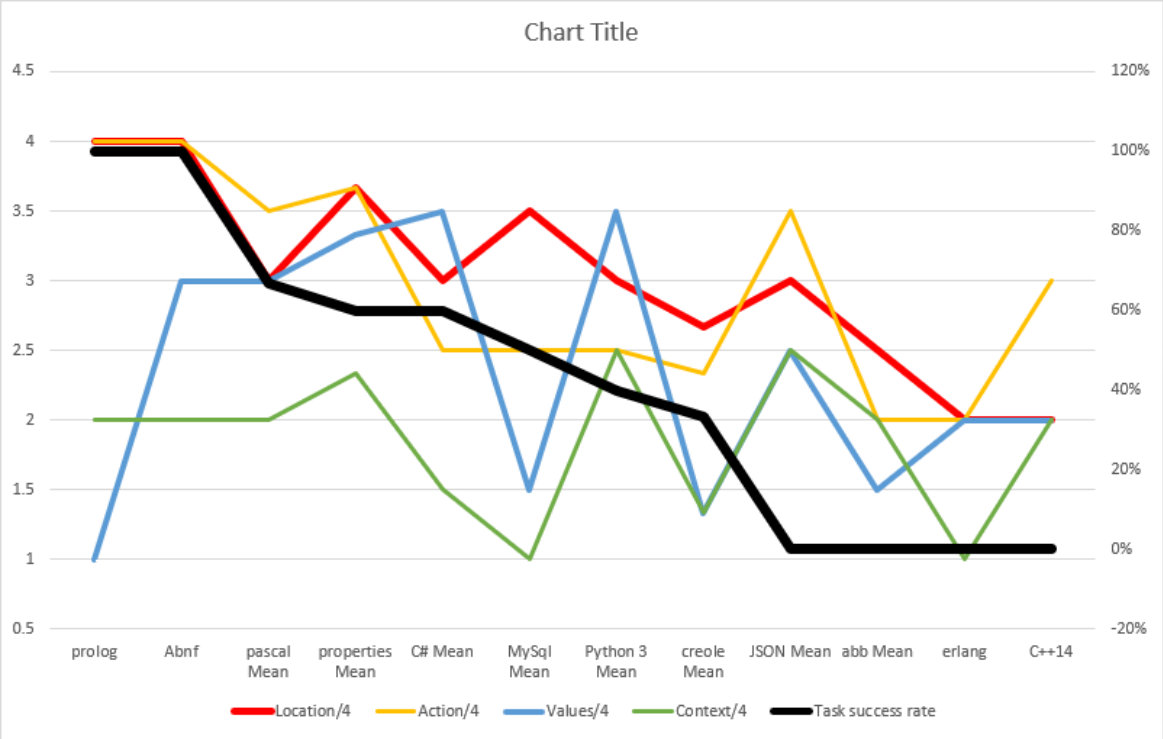

Fig. 4.5 A comparison between the success rate of activities in a programming language, and the user feedback. Only location and action vaguely trended with the activity success rate.

out of the 32 given to this sample. For the 3 participants with a recorded language familiarity score of more than 1.5, 10 tasks were successfully done out of the 17 given. A total of 41 tasks were distributed among participants, and a total of 18 were completed, while the rest were skipped.

In the following diagrams, for languages that appeared in the samples, I ordered them from highest percentage of completed tasks, to lowest. For tasks with the same percentage of completions, they were ordered from lowest number of activities given, to the highest number of activities given. This is because with less activities, there were less opportunities that could be completed.

The technical analysis legibility ratings did not correlate with the success rates of the activities. We investigate the potential reasons for this in the following subsections, breaking down each statistic and finding what correlates. We aimed to find whether there were notable enough patterns despite the sample size of 7 users, 12 occurring languages and 41 occurring tasks.

Figure 4.5 contains a graph comparing the task completion rate for languages, and the user ratings on each section of feedback for tasks of those languages. Against the scores

Language	Lower Quatile Actions	Meidan Actions	Upper Quatile Actions	Too few steps	Too few and too many	Right no of steps	Too many steps
prolog	0	0	4			XX	
Abnf	0.5	3	12.5			XX	
pascal Mean	1.25	7.5	13.75		X		X
C# Mean	1	2	3		X		X
properties Mean	0.25	2	3.25		X	XX	
MySql Mean	3.25	10	24			X	X
Python 3 Mean	2	15	18		X	X	
creole Mean	1	5	10	X	X		X
JSON Mean	11	12	14			XX	
erlang	8	9	11.5		X		
abb Mean	85	85	87		XX		
C++14	1	17	34.5		X		

Fig. 4.6 A comparison between the range 'amount of actions from the goal' for each language, and what users reported on the number of steps. Checks with 2 x's imply a consensus between multiple participants.

provided for feedback, there is a rough correlation between the success rate of tasks, and the responses provided for feedback.

- Location followed the values of the success rates rather accurately, but with the occurrence of 3 shallow (0.5 value) spikes.
- Action had an overall downward trend, but high values for JSON and C++14.
- Value consisted of many differing results, but overall had lower values matching the lower success rate, and higher values matching the higher success rate. The scores in Value consistently had a standard deviation of around 0.5.
- Context displayed no relation to the success rate, despite the values having the lowest standard deviation (with a large exception being JSON with standard deviation of 1.5). It is possible the meaning of this value was misphrased, or misunderstood. For instance, one properties feedback contained literal words such as "Line". It is possible the user stated "syntax terms" were not helpful because they believed they did not appear. Providing a qualitative string box would have been useful for letting the users convey their reasoning for these scores. The median of the standard deviation of the feedback was 0.47, so despite the unexpected results, there was a general consensus among participants.

Action Count in relation to User feedback on Step and Detail

For the number of actions generated per language, we obtained the upper quartile, median and lower quartile, illustrated in figure 4.8. These values were auto generated, and hence contained consistent scoring across languages.

I evaluated this figure against the "steps" score for each language. We did this to check the integrity of the quantitative feedback on "steps" recorded from users by the survey and

Languages	Lower Quatile Actions	Meidan Actions	Upper Quatile Actions	Needed less detail	Needed more and less	Right amount of detail	Needed more detail
prolog	0	0	4			xx	
Abnf	0.5	3	12.5			xx	
pascal Mean	1.25	7.5	13.75	x		x	
C# Mean	1	2	3	x		x	
properties Mean	0.25	2	3.25			xx	x
MySql Mean	3.25	10	24		x		x
Python 3 Mean	2	15	18			x	x
creole Mean	1	5	10	x			xx
JSON Mean	11	12	14			xx	
erlang	8	9	11.5				x
abb Mean	85	85	87				xx
C++14	1	17	34.5				x

Fig. 4.7 A comparison between the range 'amount of actions from the goal' for each language, and what users reported on the amount of detail on activities. Checks with 2 x's imply a consensus between multiple participants.

found very ambiguous results. A comparison of results against "steps" are displayed in Figure 4.6.

There was no relation to the values in Steps, and the action counts. "Too little or too many varied" occurred regardless of the number of Action Steps - occurring more than any of the others. Additionally "Too few" only occurred once, in a language where 2 other participants did not provide this response. Only one user gave the same response for Step across all their languages, meaning the response also varied even across the same participants. There appears to be no relation between how Step was selected, and the number of Actions for each language.

I also evaluated the action count in relation to the surveyed rating of "detail". In contrast to the comparison between action count and steps, there appeared to be an appropriate relation. Figure 4.7 contains a comparison of results against the "detail" ratings.

The most common response was wishing each bit of feedback contained more detail, which occurred for 7 of 11 languages - 2 of which could be recorded as 'unanimously stated'. This coincides with how users wished the instructions contained more detail in River's observed experiments with ITAP[16]. "More detail" occurred more when there were higher numbers for action counts, which also corresponded with the completion rate of activities. "Right amount of detail" occurred more with lower action counts - also corresponding with the completion rate of activities. The amount of participants wanting more detail was inversely proportional to the number of 'Actions from the goal' Selection of "Less detail" occurred for 3 of 11 languages. We can not conclude on a relation to the action counts and the rate of other detail responses.

Descriptiveness, Context and Requested Detail

There were 3 values relating to describing a phrase of code to the learner. These (each for comparing the usefulness of the generated token terms) were:

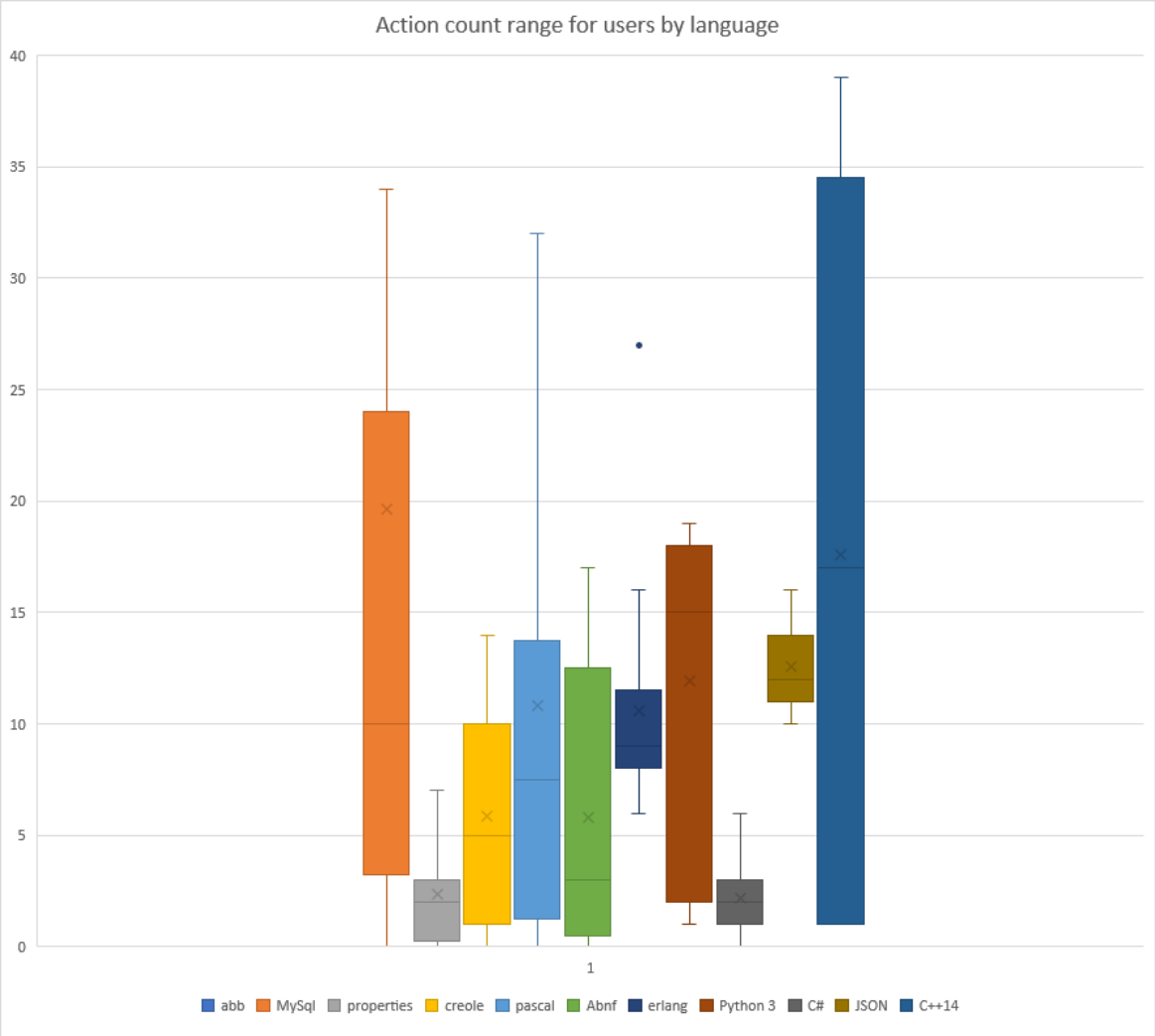

Fig. 4.8 The range of 'number of actions generated for users when they clicked check' for each language with at least 2 tasks performed across participants. abb is off the chart with values of 85

- Token "Descriptiveness" Score (Technical analysis) - This is a calculated value based on the source grammar. It is calculated using the number of descriptive (non-technical) terms and adjectives in token terms. "The total percentage of "Language (Token string) Technical Terms" + "Literal Terms (Descriptive/Adjective)" minus the percentage of "Technical" words"

- Multiple Words (Technical analysis) - This is also a calculated value based on the source grammar. It is calculated using the number of words used to form token terms (phrases).

- Context (User evaluation) - Statistic stated by the user on how much the context value (formed by the token names) helped.

- Requested Detail (User evaluation) - The user states if they wish more detail was in each feedback instruction. This could also relate to the other parameters in feedback, however it is expected to relate to the context and descriptiveness of the language as those also describe feedback.

- Discernible (User evaluation) - A more direct statistic on the phrasing of words, selected by the users. This is a general quantification on the phrasing in the feedback. Its purpose is to check a consistent relation between the descriptiveness of the tokens, and how well the users interpreted the terms.

Sql is excluded in this section as it was not evaluated in the technical analysis. The results for Discernible, Requested Detail and Descriptiveness are displayed for comparison in Figure 4.9.

The rated descriptiveness from the technical analysis appeared to weakly correlate with the requested amount of detail. 4 of 11 languages were rated with positive descriptiveness, where one of these received a "requested more detail" rating. In contrast, 7 of the 11 languages were rated with negative descriptiveness where 5 received a "requesting more detail". The number of words appeared to weakly inversely correlate with the requested amount of detail, which contrasts the expectation. 1 out of 4 languages had more detail requested in the survey, among the languages with less than or equal to 40% of grammar phrases consisting of multiple words. 5 out of 7 languages had more detail requested in the survey, among the languages with more than 40% of grammar phrases consisting of multiple words. This implies that the languages with a higher percentage of words for token names were much less likely to have users evaluate they needed more detail.

Descriptiveness appeared to have little impact on Context. Context heavily correlated with Location, Action and Value - often being a lower value than each. However, descriptiveness

Languages	NoSpaces/4	Abbreviated/4	Undiscernable/4
prolog	3	4	4
Abnf	3	4	4
pascal	1.5	3.5	3.5
C#	2	3	2.5
properties	2.666666667	2.666666667	3.333333333
MySql	1.5	3.5	2
Python 3	1	3.5	3
creole	3	1.333333333	2.666666667
JSON	3	2.5	1.5
erlang	4	1	2
abb	2.5	2	2
C++14	2	3	2

User evaluation scores for the phrasing in feedback

Illegible Tokens	Technical Tokens	Literral Tokens	Descriptive Token Score	Multiple Words
25.00%	50.00%	25.00%	-37.50%	25.00%
30.77%	30.77%	38.46%	15.38%	40.00%
8.22%	56.16%	35.62%	10.96%	80.41%
10.85%	47.29%	41.86%	13.95%	113.81%
14.29%	42.86%	42.86%	-14.29%	16.67%
x	x	x	x	x
30.00%	52.86%	17.14%	-5.71%	80.00%
23.08%	38.46%	38.46%	-30.77%	46.15%
28.57%	42.86%	28.57%	-28.57%	0.00%
49.23%	35.38%	15.38%	-12.31%	82.11%
7.14%	57.14%	35.71%	-7.14%	50.00%
26.14%	44.32%	29.55%	9.09%	91.15%

Technical analysis scores for the tokens in grammar files

Fig. 4.9 For each grammar, the scores for the user's observations, compared to the technical analysis' observations on each language's 'descriptiveness'

did correlate with the values in Abbreviated and Discernable. Of the 7 languages with negative descriptiveness, 4 had discernable scores lower than 3. The 3 exceptions to correlating values between descriptiveness and the user evaluation scores were prolog, properties and Python 3. prolog had high abbreviation and discernible scores, yet incredibly low descriptiveness and token legibility scores. However, prolog had only a single instance of being attempted, and was performed by a user who rated an exceptionally high familiarity score of 5 - providing 2 possible reasons for this exception. properties' grammar has an exceptionally low number of illegible tokens, and the highest number of literal tokens. Python 3 had the highest negative discernable rating, and a Discernable score of exactly 3, and the second highest standard deviation for discernible scores at 1.

2 out of 4 of the languages with positive descriptiveness scores had a discernable rating of 3 or higher. C# and C++14 had discernable ratings of less than 3. C++14 had low literal tokens and a user rated abbreviation value higher than 3. C# was also an exception with 2.5, and a standard deviation of only 0.5. This is a particularly notable exception as its values were ranked as Explicit, containing:

- a low number of illegible tokens,
- a high number of literal tokens,
- a high descriptive token score,
- an erroneous number of tokens with multiple words.

The number of "literal tokens" recurred as a potential reason for differences between the discernable score, and descriptiveness score. However, when comparing the two values, the results are scattered.

Overall, the trends are loose between the technical analysis legibility scores and the user evaluated scores. However it is arguably significant that each compared statistic consistently

has more or equal positive values with positive values, and more or equal negative values with negative values.

Summary of RA2b

A lot of the connections are loose between the technical analysis legibility scores and the user evaluation. Of the values that our research noted as potential factors that could affect legibility, "descriptiveness" and "literal tokens" had a stronger relation. The impact of context in general was consistently lower than Location, Value and Action. This may imply the other 3 key terms in adaptive feedback have a larger prevalence, which is worth investigating. We can only vaguely conclude on the legibility of context is based on these results alone. Legible tokens appeared to benefit the how legible the hints were. Matching the low count of programming language familiarity, primarily "Technical" legible languages had were lower. We can conclude that word count had no affect on the legibility. Two thorough test are needed in isolation from one another. It's unclear how much context impacts adaptive feedback. It's also unclear what components affect token legibility.

According to the technical analysis alone, the median of languages are "Technical", which may be not ideal for users unfamiliar with programming. This is not-ideal as IPT exist specifically for a range of users such as those unfamiliar with programming. However, the impact of this may be negligible if the usefulness of context (relative to the rest of the adaptive feedback) is as small as implied.

4.2.2 RA3a - What conditions in grammar definitions can be listed/categorised to affect the speed feedback can be returned? How does the compile speed change for different contexts?

The build failed to include a timer for the compilation time of each check call. This would be checked against the language and the amount of text for each compilation.

However, the technical analysis made notes on each given language for the maximum delays observed when a call to generate a solution was made. The results of this are noted in Table 4.2.

We can not definitively answer 'What conditions in grammar definitions can be listed/categorised to affect the speed feedback can be returned? How does the compile speed change for different contexts?'. However, from the technical analysis, we note that while time rises exponentially with the goal and problem size, it is not a definitive factor. There also appears to be no relation between the compilation speed, and the size of the code within the grammar file to perform tokenising and parsing.

Language	Grammar total lines	Maximum goal line count	Maximum goal character count	Maximum observed Delay
abb	345	12	337	Instant
abnf	111	55	1877	20 seconds
bcpl	131	389	11380	5 seconds
C++	1212	30	814	Instant
creole	135	118	3091	Instant
C#	2308	1072	30783	5 seconds
erlang	432	64	1640	Instant
json	76	22	583	Instant
Lua	281	13	255	Instant
pascal	858	487	19472	Instant
prolog	149	5	79	Instant
properties	51	122	3896	Instant
python	577	1468	56453	15 seconds
toml	115	242	5233	2 seconds
webidl	711	228	12643	10 seconds

Table 4.2 The maximum observed delays during the technical analysis. The maximum character cound and line count are obtained from the pool of examples used, however the clipboard was capped at approximately 8192 characters.

How this comment was parsed	Occurrences
"A name of a definition contains ‘comment’, ‘COMMENT’ and so on." (parsed or tokenised)	3
"A definition is linked to a ‘skip’ command."	7
"A definition is linked to a ‘channel(HIDDEN)’ command."	6
"A definiton is linked to a ‘channel(X)’ command. Moreover, X contains ‘comment’, ‘COMMENT’, and so on"	3

Table 4.3 The number of occurrences of the different methods grammars use code comments.

This is open to further research.

4.2.3 RA3b - What is the frequency of grammars including, streaming or excluding (skipping) code comments?

In this section, we discuss, ’What is the frequency of grammars including, streaming or excluding (skipping) code comments?’. The exact ways each of the sampled grammars stored code comments is recorded in Figure 4.3.

Semura’s work noted 4 types of comments [43] (although the quantity of each was not mentioned). Our survey observations matched those patterns. The numbers of each within our samples were as follows in Table 4.3.

We also noticed occasions where no comments were identified, or comments were parsed normally as a token. 4 grammars of this research’s did not handle code comments.

Two thirds (66.7%) of comments are attempted to be obscured - skipped or hidden - in the languages that contain code comments. In the case of accessing separate channels, comments channel would successfully provide the information that can be pulled. In contrast, the ’Hidden’ channel is often a substitute for all actions skipped, and would be conflated with characters such as spaces between tokens (depending on the language).

In instances where comments are parsed they can simply be retrieved, however the solution will require their addition and exact phrasing, unless they are a normalised parameter.

It would be ideal if grammars were written with comment usage in mind, but to circumvent comment skips, the grammar would have to be modified. In the majority of programming languages, the code comment token was literal. Hence fixing this would be a process of finding the token and changing it to a specific channel, or having it parsed. To have this

solution be domain-specific, the ideal would be to stream it to a shared channel name for comments across grammar files such as "Comment".

Summary of RA3b

The majority of code comments are skipped, however they are often literally named and a swap of function can be performed. If the comments are streamed to the same channel, this creates the possibility of being able to identify the inclusion of code comments, read code comments, respond to their position (or lack of position) and to use code comments in feedback.

4.3 Chapter Summary

Through this chapter, we first used the technical analysis to evaluate aims: 1a, 1b and 2a. Out of all the included grammars, 99% of problem code tested in Nue Tutor returned appropriate steps to reach a given goal code from problem code. However, despite sharing the same weighting aims as ITAP[17], the method at the time unreliably identified which goal code was closest to the problem code out of a selection of goal codes. Only 40% of grammars correctly identified the correct closest goal codes reliably. This is attributed to being an older version of the weighting algorithm and needs re-evaluation. Overall, we declared a success for the use of the algorithm for language-independent adaptive feedback.

For language-specific components, we listed one must facilitate for each language:

- Finding the correct root token.
- Multiple compilations. (For language exceptions - 1 out of our 16 included)
- Which tokens should be normalised. (Functional without)
- Compilation into a Virtual Machine. (Functional without)
- Quality Feedback - (Functional without)
- Layout Feedback - (Functional without)

As for which token to use for the root token, we found it was reliable to use a token not included in any phrase, and with a definitively ultimate name such as: document, file, or the language name. This was at a rate of 75% of the sampled languages. It is still critical to verify and find the appropriate root token (or tokens in the rare case of multiple compilations).

We then used the user evaluation to evaluate aims 2b, 3a and 3b. The integrity of the user evaluation results was questionable, but patterns between the scoring within the technical analysis and the user evaluation were found. The number of literal terms within tokens, and the number of descriptive terms within a token affected how easy to determine the meaning of each token was. The number of words and technical terms did not affect how legible the tokens were. The overall scored experience of participants using programming languages at all was very low, which may be the reason technical terms had no definitive occasions where they could be observed to help. Out of the 4 parameters forming feedback (Location, Value, Action and Context) Location and Value appeared to help considerably, and Context was relatively lower regardless of descriptive score.

Compilation of ANTLR4 was consistently below 20 seconds, with a median of "Instant" compilation speed. The amount of characters in code affected the compilation speed, but the number of lines forming the language grammar did not.

Finally, we noted the pattern with how grammars treated code comments was consistent with Semura's work [43]. The majority of code comments were skipped or hidden.

Chapter 5

Research Contributions

This research contributes in the field of Intelligent Tutoring Systems for Programming (Intelligent Programming Tutors), following unique features:

Contribution 1

Concerning the problem of IPT syntax feedback and semantic feedback having to be built very specifically for a given programming language:

We aimed to "successfully propose, describe and test a state of the art method of providing guaranteed syntax feedback and semantic feedback in IPT, for theoretically every programming language".

Through literature reviews, I found an absence of IPT that can facilitate more than one programming language. Hence I also observed that only one domain-independent and language-independent IPT existed with state of the art adaptive feedback that could return feedback regardless of the code state - for Python3 only.

For this, I documented, created and tested the method of creating the first programming language-independent IPT framework, and it is capable of hint generation; returning syntax and semantic feedback for a unanimous (greater than 99%) number of user submitted code states across programming languages.

I verified this in the IPT framework using an open source repository of code examples, and programming language grammar files not specifically built for this domain. Additionally, a user friendly build of this IPT was distributed to users unfamiliar with the framework,

Contribution 2

Concerning the problem of IPT having authoring tools needing to be made specifically for both the formatting of task creation, and the given language:

We aimed to "facilitate task creation through an authoring tool that is capable of working across programming languages".

Through literature reviews, I found authoring tools specifically for their domain in primary literature, and reviews of authoring tools through the field of IPT evaluating on a need for more domain independent applications of authoring tools [56, 58] in secondary literature.

For a programming language-independent IPT framework, I created, tested and documented the method of creating an authoring tool, that produces tasks capable of hint generation; reliably returning syntax and semantic feedback for a unanimous (greater than 99%) number of user submitted code states.

My research is not the first to use abstract syntax trees comparisons to guarantee hints are generated for any submitted code (Kelly Rivers[35] was the first to ensure the method worked reliably for all code for Python3). However, the way "canonicalisation", "path construction" and "hint generation" have been implemented is unique (reification was omitted and is an area open to research). Furthermore, the assurance and testing of the general method for all possible programming languages is unique, as well as the facilitation of working with an indefinite number of programming languages. This is the first programming language-independent IPT framework.

Contribution 3

Concerning finding a renowned repository that can be used for program synthesis which was only domain-specific to programming languages:

We aimed to "record the effects of using ANTLR4 grammar terms on the legibility of syntax and semantic feedback, in a language independent IPT".

Through my literature reviews, I found no documented discovery or application of language-independent program synthesis for IPT (or any language-independent solution for IPT). I also found no documented discussion on the naming used within ANTLR4 grammars.

I have documented, implemented and tested the first IPT that refers to a programming synthesis framework by abstraction. I tied this program synthesis framework to a renowned language (ANTLR4) to import or create languages. This ultimately makes the first solution towards domain-independent program synthesis.

ANTLR4 runtime, ANTLR4 tool and the ANTLR4 repository are not the product of this research (Terrence Parr[45] produced ANTLR4). However producing an IPT framework that can utilise ANTLR4 (or any tool linking directly to a BNF adjacent format) to produce language independent adaptive feedback is unique to this research.

5.1 Research Journey

A large motivating factor for starting this research was to provide an accessible means of providing tutoring for programming (games programming or computer science) that was personalised and responsive to each learner. I had high expectations on what had already been accomplished, but struggled to find a named term for the field. I settled on the terms 'interactive media', to 'intelligent tutoring systems' to 'cognitive tutors' and finally 'intelligent programming tutors'.

I primarily wanted to investigate or introduce what was ultimately, using PDDL[83] for a learner model attached to an authoring tool. I also wanted to investigate or introduce the effects of making such a learner model "explainable"[53, 52] or an "OLM"[50] - where both terms were ultimately synonymous with each other. I mainly wanted to investigate whether an open learner model would improve motivation with the learners using an CPT, and the accuracy of the suggested tasks.

I investigated what had been performed in IPT through literature reviews on IPT, each lessening in scope. Along the way, I struggled to find synonymous terms across IPT. I found the recurring term "adaptive feedback", as well as references to the 4 models but with inconsistently referenced sources: "learner model", "interactive model", "pedagogical model" and "domain model". My difficulty finding recurring synonyms was consistent with my findings in the final literature review I performed; there is little consensus on even the synonyms for 'ITS for programming' outside of adaptive feedback, interactive model and domain model.

I also could not find any form of "generative learning"[40] in my studies of existing IPT aside from self-explanation[84] facilitated outside Garcia's work [42]. The primary aim of my research was to introduce and investigate the effects of promoting self explanation within the IPT framework itself - through learners writing code comments. The idea was to investigate this alongside the state of the art method for providing adaptive feedback and a given learner model.

I additionally could not find any existing CPT in my own secondary study, or a thorough study of other secondary studies (documented in Section 2.1). The final problem was finding that the state of the art methods were domain-specific implementations. This became the first key investigation I intended to write up after a number of reconsidered[1] literature reviews.

I pursued finding a means to create and interface a compiler for languages I had interest in creating this for: C#, C++ and Lua. In the pursuit to perform this in the same manner across these 3 languages - and find constants across all programming languages (by definition) - I

[1] Following finding that the there was not as much (or any) content exploring for each hypothesis/research theme

found ANTLR4[46]. It not only contained a means of compiling code within Nue Tutor's built in framework, but had already solved the queries on 'what concepts were consistent across programming languages (by definition of programming languages), and could be built around'[45]. ANTLR4 runtime's process contained what ITAP[17] had to call directly from the Python3 framework, but for any given languages.

I confirmed this was completely new and unexplored - the current (at the time) state of the art system of adaptive feedback that could be used across programming languages. My research aimed to introduce and investigate the extent in which this could be used. Initially this was to serve as the foundation for my investigations with the learner model in IPT, and investigations with the self explanation model in IPT. However, due to time constraints and the fact this was a large area of investigation, this became the focus of the research.

To verify if this was genuinely possible, I started by creating an adaptive feedback framework, with ANTLR4 for program synthesis that was referred to by abstraction. I named it "*Nue* Tutor", as it could take the any form (of language) but was the same construct, method and software regardless of the form it assumed. The test interface started simply as an area to type the problem code state, an area to type the goal code state, a button to generate a solution, and an area with the generated identified solution (with each step phrased like adaptive feedback). I used a custom created tutorial language, and a few samples from the ANTLR4 repository. After bug fixing and refining, the resulting construction of its method is mentioned in Section 3.1.

I noted the potential constraints the grammars could provide, and themes to investigate regarding how well the method - language-independent adaptive feedback generation - worked. I constructed an authoring tool (with the initial research aims[2] still in mind) that could save multiple goals, and hide them from the user when loaded as a task to do. I additionally sampled more languages from the repository under a specified criteria. This state of Nue Tutor was used to perform the technical analysis, however Nue Tutor's method of hint generation was further refined in future (transfer match was excluded at the time). This was the second key investigation I intended to note alongside the first.

From this technical analysis, we confirmed the unanimous success of feedback returned (regardless of the learner's code), given a language that uses only one root note that encapsulates all note types. We also confirmed that use with unedited grammar samples from the ANTLR4 repository worked, given that they could compile into C# in the ANTLR4 tool. Additionally, we confirmed that even if the goal code has what would be syntax errors, Nue Tutor will note them and still identify a solution.

[2] "Using [PDDL][83] for a learner model attached to an authoring tool"

By this point, the focus of the research became on the language-independent aspects of the IPT: the rate of success, the phrasing of the adaptive feedback produced, and how the grammar samples affect Nue Tutor. While the framework "using planning domain definition language[83] for a learner model attached to an authoring tool" was successfully attached to Nue Tutor, and my preceding research[85] applied a method of choosing adaptive activities from these variables, I did not investigate this during this research. However, we included in our investigation the variables required for a follow up with an investigation on the use of code comments in IPT. The follow up investigation has yet to be commenced.

To verify the integrity of our technical analysis, I improved Nue Tutor, and created a separate build for the user evaluation. A test case for the build was distributed to one participant, however due to time constraints and late feedback received from the build, it was sent to all the other participants in that form. This build did not provide the slot for qualitative feedback (to verify if the meaning of each survey question term was interpreted correctly, or additional thoughts) and did not contain the time between each click of "check" and feedback rendering. The interface would also not expand according to screen size (potentially hiding text). However, everything else was functional. The user evaluation raised more questions than answers, and the integrity of the user analysis itself is admittedly questionable with 7 participants. However, there was a general consensus among the quantitative feedback returned by the participants that was used to note patterns in what part of the instructions they found useful, and the effectiveness of literal token names (compared to all other forms of 'descriptiveness' we noted).

This formed the third key investigation and the final investigation for this thesis. The three investigations were compiled into this thesis, documenting our contribution to knowledge on "an adaptive feedback framework for a language-independent intelligent programming tutor (IPT) using ANTLR4". This concludes this research, however I still personally will use the knowledge gained in this research to pursue the initial research questions indefinitely in future.

5.2 Conclusion

ITS for programming (which can also be referred to as IPT), is a less explored and defined sub-field of ITS. The field is overall inconsistently defined, however there is a prevalence on the importance of adaptive feedback in IPT.

Very few IPT share methods, or the programming languages they're built for. IPT are also considered functional for very specific programming languages. Hence, the tools made to create tasks (authoring tools) are also considered domain-specific.

Given text for code, to propose, investigate and test the first domain-independent adaptive feedback system for IPT, I successfully proposed, documented and investigated an IPT that:

- Refers to programming languages by abstraction - as simply abstract syntax trees.
- Performs program synthesis by abstraction using ANTLR4, with a renowned programming language for part of the process of converting programming languages to syntax trees.

We performed a technical analysis with 16 sampled languages from the ANTLR4 repository. From the initial sample of 23, we excluded 7 as not all grammars compiled. Each included grammar functioned with the authoring tool and for generating tasks. Feedback was produced more than 98% of the time across all submitted learner code states. The quality of feedback produced and the accuracy of the trees varied.

For the feedback, the location, value, action and context all functioned. The phrasing of the actions, location and actions reportedly helped with completed tasks, yet the phrasing can be further developed. Value reportedly helped more or less with no recognisable pattern. However, context was consistently marked low as rarely helpful, with no relation to their expected technical analysis descriptive score. It is worth investigating whether this is the perceived usefulness of context relative to the other statistics, a result of implemented ordering and phrasing, or an affect of the phrasing of the grammar files.

Within feedback, Location and Value are strict unedited statistics from the code, making these difficult to rephrase or improve. Clarifying what the value refers to or consist of, or what the location is (and how to find it) are variables to consider.

In contrast, the Context and Actions within feedback can be heavily affected by phrasing. Context is pulled directly from grammar files, and hence improvement of context would lie in the improvement of grammar files.

As context had no correlation on the results of the user evaluation, we can not conclude on the magnitude of what phrasing benefits context. Additionally, the sample size consisted of 7 different users. However, we identified observed patterns for what can be considered to affect context:

- The number of technical words in contrast to the number of literal words roughly correlated with how reportedly easy it was to know what context referred to.
- The number of words in the context did not appear to correlate with how reportedly easy it was to know what context referred to.
- The number of spaces in the context did not appear to correlate with how reportedly easy it was to know what context referred to.

- The number of abbreviations, correlated with how easy it was to discern what context words themselves meant.
- The number of abbreviations also roughly correlated with how reportedly easy it was to know what context referred to in code, as a likely result of the former.

There is potential for further research into how each component of this structure of adaptive feedback (Location, Action, Value, Context) helps feedback. There is also potential research into what can improve the helpfulness of context not just in Grammar files, but in adaptive feedback in general.

Finally, we found that for further research to make use of ANTLR4 grammars for code comments, there is a likelihood one may have to modify the grammar of particular languages to support including code comments. This is because the majority of grammars omit code comments. This means not all grammars would be compatible with any feature that uses code comments. However it would be worth pursuing the inclusion of reading code comments to facilitate and test the affects of self explanation. The implementation could append that feature if comments are supported in the correct manner, and function without otherwise.

To pursue an investigation on how self explanation in such an IPT would affect the quality of learning, this IPT needs to be built and have tasks added for an educational setting, run and then evaluated.

We wish to further develop this research and methods, using the IPT and developing it into a cognitive tutor that ties a language-independent authoring tool to a learner model and the recommended tasks. Additionally we wish to develop the IPT to use code comments for self explanation (in the adaptive feedback framework). Furthermore, we wish to also use code comments for manually phrased code hints in the authoring tools framework. Each of these will be implemented and tested for all programming languages through this singular domain-independent IPT, or any other research resulting from (now defined) domain-independent IPT.

Glossary

This section contains a list of all abbreviated terms used throughout this thesis. Each entry has the abbreviated term, their full phrasing, and their definition.

	Phrase	Description
	Authoring Tool	A built in tool to an IPT that helps/facilitates the construction of learning activities by tutors. These learning activities are then supplied by the IPT for learners to complete.
AAT	Automated Assessment Tools	Used in the context of programming. Websites, software or computer tools that provide programming tasks, then evaluate the success of those tasks.
ALL	Adaptive LL	An algorithm for tokenising and parsing. This algorithm is an adaptation of "LL(*)" to enable the parsing of grammars to identify ambiguities dynamically at parse time (rather than statically defined by the construction of the grammar).
AST	Abstract Syntax Tree	A full representation of what refers to what within code. Contains nodes referring to tokens. These nodes branch off into definitions for other nodes, forming a literal tree of the syntax.
BNF	Backus Naur Format	A popular writing convention for defining the syntactic rules of a programming; a "meta-language".
	Built in	The used definition of this phrase is: any feature that does not require recompilation of the tool to use.
	Class 3 Definedness	For a given learning activities, these have *multiple solution strategies* that can be used to determine if the task is successfully completed.

	Phrase	Description
EBNF	Extended Backus Naur Format	Contains a few formatting differences compared to BNF, and facilitates the ability to handle "repetition of syntactic rules", "special sequences", "optional choice of syntactic rules" and "exceptions to syntactic rules".
	Explainable	In the context of AI, refers to "XAI". Ultimately, the act of making an AI decision transparent and interactive. Transparent by having AI able to communicate how it came to a decision. Interactive by having AI able to accept correction.
IPT	Intelligent Programming Tutor	ITS for programming. Websites, software or computer tools that provide programming tasks, or information. Rather than just verifying how correctly a task is completed, they provide contextual feedback, hints, and optionally suggestions on content to navigate.
ITAP	Intelligent Teaching Assistant for Programming	A state of the art IPT for Python 3. Capable of returning hints regardless of what code is written (relative to a task). Additionally, supports using correct learner code to further develop the solutions it targets, and the hints it provides.
ITS	Intelligent Tutoring System	Automated systems designed to provide adaptive, or personalised learning for a given subject.
LL	LL	A "top down" parsing algorithm. The name is not an acronym.
OLM	Open Learner Model	For systems that store information on the user (to adjust content accordingly), these models enable learners to inquire and modify this information to some capacity.
	Parsing	The process of forming structures or phrases from a series of tokens according to the rules of a given programming language. This process forms an abstract syntax tree (AST) - a full representation of what refers to what within code.
PDDL	Planning Domain Definition Language	A language for state space planning. Allows for definitions of a "world state", objects within a world, actions that can change the world, required states for actions, initial "problem" states of the world and targeted "goal" states of the world. Algorithms can then iterate to find a combination of actions that get from the problem state to a goal state.
	Program Synthesis	The process of identifying a program within a programming language, grammar or high level specification.

	Phrase	Description
NLP	Natural language processing	Research around having computers parse natural sentences and potentially respond with natural sentences.
RA#	Research Aim	Critical questions this research aims to answer. Has a number and a sub-letter.
RO#	Research Objective	Methods to answer research aims of the corresponding number and sub-letter.
	Self Explanation	The process of a learner redefining a concept. Reciting, listing, restructuring through notes, rewriting or phrasing their definition.
	Semantic Feedback	Feedback on how a learner can reach a goal.
	Semantic Variant	Grammars of the same version of the same language, but written differently
	Significant Variant	Grammars of a different version of the same language.
	Syntax Feedback	Feedback reporting oddities in syntax and how to fix those.
	Token	Recognisable combinations of letters, numbers symbols or formatting terms forming a singular 'word' called a "token".
	Tokenizing (Lexing)	Also referred to as lexing, it is the process of identifying "tokens" from a series of letters, numbers, symbols or formatting terms. Converts a series of text characters into legible words for a given programming language's rules.

References

[1] N.-T. Le, "A classification of adaptive feedback in educational systems for programming," *Systems*, vol. 4, no. 2, p. 22, 2016. [Online]. Available: https://www.mdpi.com/2079-8954/4/2/22/htmhttps://www.mdpi.com/2079-8954/4/2/22

[2] A. Kyrilov and D. C. Noelle, "Binary instant feedback on programming exercises can reduce student engagement and promote cheating," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 2015, pp. 122–126. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2828959.2828968?casa_token=GCKasckjuNgAAAAA:93rf9rUSz8o9S8EQmA9kVdqEPr1t8fO9Et0Bo_QINSyCrhW7qgPY-LnDujFf4CJxzZxe9bvFqdUN

[3] M. E. Caspersen and J. Bennedsen, "Instructional design of a programming course: a learning theoretic approach," in *Proceedings of the third international workshop on Computing education research*, 2007, pp. 111–122. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1288580.1288595?casa_token=RmKecSWUAQ0AAAAA:To-P6dd71x84LO2_sAWNXFSkcf0AbT_xLgnH7RGRjkuz2TB0BTbA7kUvRVIxtxmFOPVqD3qrJ5VE

[4] C. Watson and F. W. B. Li, "Failure rates in introductory programming revisited," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 2014, pp. 39–44. [Online]. Available: https://dl.acm.org/citation.cfm?id=2591749

[5] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming," *AcM SIGcSE Bulletin*, vol. 39, no. 2, pp. 32–36, 2007.

[6] J. Carter, P. Dewan, and M. Pichiliani, "Towards incremental separation of surmountable and insurmountable programming difficulties," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 2015, pp. 241–246. [Online]. Available: https://dl.acm.org/citation.cfm?id=2677294

[7] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys (CSUR)*, vol. 37, no. 2, pp. 83–137, 2005. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1089733.1089734?casa_token=T5XMwJ--xrEAAAAA:ueqTarjvkCZOKnMfNeRD4MRJN_Pd6e3-GR2uWUM7CzCTnYyT-l1ipW-igK4uRthdl6YrPydan6tw

[8] Vermunt Jan D., , and V. Donche, "A Learning Patterns Perspective on Student Learning in Higher Education: State of the Art and Moving Forward," *Educational*

Psychology Review, vol. 29, no. 2, pp. 269–299, 4 2017. [Online]. Available: https://link.springer.com/article/10.1007/s10648-017-9414-6

[9] H. Keuning, J. Jeuring, and B. Heeren, "Towards a Systematic Review of Automated Feedback Generation for Programming Exercises–Extended Version," *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education. 201*, p. 18, 2016. [Online]. Available: http://www.cs.uu.nl/research/techreps/repo/CS-2016/2016-001.pdf

[10] T. Crow, A. Luxton-Reilly, and B. Wuensche, "Intelligent tutoring systems for programming education: a systematic review," in *Proceedings of the 20th Australasian Computing Education Conference*. ACM, 2018, pp. 53–62. [Online]. Available: https://dl.acm.org/citation.cfm?id=3160492

[11] E. Mousavinasab, N. Zarifsanaiey, S. R. Niakan Kalhori, M. Rakhshan, L. Keikha, and M. Ghazi Saeedi, "Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods," *Interactive Learning Environments*, vol. 29, no. 1, pp. 142–163, 2021. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/10494820.2018.1558257?casa_token=IYdUNCMD2yIAAAAA%3AilhHZ8X-oEX6ZKHeD58TlCjhNYdMa7rmn__C5A6MgQhtzUB9qx1zDm5WHFS7jtByzMQPE0ZSVRw

[12] K.-Y. Tang, C.-Y. Chang, and G.-J. Hwang, "Trends in artificial intelligence-supported e-learning: a systematic review and co-citation network analysis (1998–2019)," *Interactive Learning Environments*, pp. 1–19, 2021.

[13] J. A. Kulik and J. D. Fletcher, "Effectiveness of intelligent tutoring systems: a meta-analytic review," *Review of educational research*, vol. 86, no. 1, pp. 42–78, 2016. [Online]. Available: https://journals.sagepub.com/doi/abs/10.3102/0034654315581420?casa_token=gEwzbC4icvkAAAAA:6McUtx_BLEwOaScjNM_3ILbMvZvF4XBOtvpGg1Cv8g0bKJ8dIStOpd8hnZLUA8kG47GOmokOoLWE

[14] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, pp. 4–es, 2005. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1163405.1163409?casa_token=L-33JRpQaK0AAAAA:KAyVFhU3Vj-UdhnCmpsckjDj2CT4krYtDPkKz-DrXt5O69lgLegnBA1QM6LMICkvOd28VKQiZ7m7

[15] R. Pettit and J. Prather, "Automated assessment tools: Too many cooks, not enough collaboration," *Journal of Computing Sciences in Colleges*, vol. 32, no. 4, pp. 113–121, 2017. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3055338.3079060

[16] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: a self-improving python programming tutor," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 37–64, 2017. [Online]. Available: https://www.researchgate.net/profile/Kenneth_Koedinger2/publication/283468835_Data-Driven_Hint_Generation_in_Vast_Solution_Spaces_a_Self-Improving_Python_Programming_Tutor/links/5702a59e08ae646a9da8771b.pdf

[17] Kelly Rivers, "Automated Data-Driven Hint Generation for Learning Programming," *Carnegie Mellon University ProQuest Dissertations Publishing*, 2017. [Online]. Available: http://krivers.net/files/thesis.pdf

[18] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 65–100, 2017. [Online]. Available: https://pure.royalholloway.ac.uk/portal/files/25913698/compilation.pdf

[19] J. Yoo, C. Pettey, S. Yoo, J. Hankins, C. Li, and S. Seo, "Intelligent tutoring system for CS-I and II laboratory," in *Proceedings of the 44th annual Southeast regional conference*. ACM, 2006, pp. 146–151. [Online]. Available: https://dl.acm.org/citation.cfm?id=1185482

[20] G. Weber and P. Brusilovsky, "ELM-ART: An adaptive versatile system for Web-based instruction," *International Journal of Artificial Intelligence in Education (IJAIED)*, vol. 12, pp. 351–384, 2001. [Online]. Available: https://telearn.archives-ouvertes.fr/hal-00197328/

[21] A. Vizcaíno, J. Contreras, J. Favela, and M. Prieto, "An adaptive, collaborative environment to develop good habits in programming," in *International Conference on Intelligent Tutoring Systems*. Springer, 2000, pp. 262–271. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45108-0_30https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.590.7140&rep=rep1&type=pdf

[22] C. Kalina and K. C. Powell, "Cognitive and social constructivism: Developing tools for an effective classroom," *Education*, vol. 130, no. 2, pp. 241–250, 2009. [Online]. Available: https://docdrop.org/static/drop-pdf/Powell-and-Kalina-U6g4p.pdf

[23] A. Vizcaíno, "A simulated student can improve collaborative learning," *International Journal of Artificial Intelligence in Education*, vol. 15, no. 1, pp. 3–40, 2005. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.372.7551&rep=rep1&type=pdf

[24] S. Marwan, J. J. Williams, and T. Price, "An evaluation of the impact of automated programming hints on performance and learning," in *ICER 2019 - Proceedings of the 2019 ACM Conference on International Computing Education Research*, 2019.

[25] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Transactions on Computing Education (TOCE)*, vol. 19, no. 1, pp. 1–43, 2018. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3231711?casa_token=hMtn_jxkOFQAAAAA:N0V6DTuZaqV4s3Jr36dEQSMAUpzXy-IesIzMcnRpWmWc7upL91kZ0fe-nCbKUq_OsVrGzGrro6Y

[26] J. C. Nesbit, O. O. Adesope, Q. Liu, and W. Ma, "How Effective are Intelligent Tutoring Systems in Computer Science Education?" in *2014 IEEE 14th International Conference on Advanced Learning Technologies*. IEEE, 2014, pp. 99–103. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6901409?casa_token=lWhB33OIOYMAAAAA:WWv6I8BXnBZeUGkRg-Orp7a8bu8cpAteqb7lwilcw6QKhyMJ-qmJzd9K0a1s4ATGcOk0TqdXZnU

[27] O. Zawacki-Richter, V. I. Marín, M. Bond, and F. Gouverneur, "Systematic review of research on artificial intelligence applications in higher education–where are the educators?" *International Journal of Educational Technology in Higher Education*, vol. 16, no. 1, pp. 1–27, 2019. [Online]. Available: https://educationaltechnologyjournal.springeropen.com/articles/10.1186/s41239-019-0171-0?fbclid=IwAR0vSk4s9y0V0vExpcAel6yL4LEb-PrNDnlreOB5WrGxlu8-3awpYGgK6Ig

[28] J. M. Harley and R. Azevedo, "Toward a feature-driven understanding of students' emotions during interactions with agent-based learning environments: A selective review," *International Journal of Gaming and Computer-Mediated Simulations (IJGCMS)*, vol. 6, no. 3, pp. 17–34, 2014. [Online]. Available: https://www.researchgate.net/profile/Jason-Harley/publication/273707429_Toward_a_Feature-Driven_Understanding_of_Students'_Emotions_during_Interactions_with_Agent-Based_Learning_Environments_A_Selective_Review/links/56041bd008ae8e08c0897e52/Toward-a-Featu

[29] Y. Fang, Z. Ren, X. Hu, and A. C. Graesser, "A meta-analysis of the effectiveness of ALEKS on learning," *Educational Psychology*, vol. 39, no. 10, pp. 1278–1292, 2019. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/01443410.2018.1495829?casa_token=sR96yCrG1-QAAAAA%3AbVuRX8rRtKPEeqYIS8GMJMhJX2U16tks5RGfXkiprORuceh1PvrAn_vcQs-Vqzygrp9uEE96Bms

[30] K. R. Koedinger and V. Aleven, "Exploring the assistance dilemma in experiments with cognitive tutors," *Educational Psychology Review*, vol. 19, no. 3, pp. 239–264, 2007. [Online]. Available: http://neuron.csie.ntust.edu.tw/homework/98/NN/KDDCUP2010/References/ExploringtheAssistanceDilemmainExperimentswith.pdf

[31] A. Arguel, L. Lockyer, G. Kennedy, J. M. Lodge, and M. Pachman, "Seeking optimal confusion: a review on epistemic emotion management in interactive digital learning environments," *Interactive Learning Environments*, vol. 27, no. 2, pp. 200–210, 2019. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/10494820.2018.1457544?casa_token=YY8Blgxdq5sAAAAA%3Auc8DRjev7eGDK562r_wruuoS8dTey3tKVtNcAFnNxENqkN-tMwRhmDudchjhkl6XsIFwAaybAW4

[32] N.-T. Le, F. Loll, and N. Pinkwart, "Operationalizing the continuum between well-defined and ill-defined problems for educational technology," *IEEE Transactions on Learning Technologies*, vol. 6, no. 3, pp. 258–270, 2013. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6497037

[33] C. Piech, M. Sahami, J. Huang, and L. Guibas, "Autonomously generating hints by inferring problem solving policies," in *Proceedings of the second (2015) acm conference on learning@ scale*, 2015, pp. 195–204.

[34] T. W. Price and T. Barnes, "An Exploration of Data-Driven Hint Generation in an Open-Ended Programming Problem." in *EDM (Workshops)*, 2015.

[35] K. Rivers and K. R. Koedinger, "Automating hint generation with solution space path construction," in *International Conference on Intelligent Tutoring Systems*. Springer, 2014, pp. 329–339. [Online]. Available: http://www.krivers.net/files/its2014-paper.pdf

[36] R. Suzuki, G. Soares, E. Glassman, A. Head, L. D'Antoni, and B. Hartmann, "Exploring the design space of automatically synthesized hints for introductory programming assignments," in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2017, pp. 2951–2958. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3027063.3053187

[37] D. Lavbič, T. Matek, and A. Zrnec, "Recommender system for learning SQL using hints," *Interactive Learning Environments*, vol. 25, no. 8, pp. 1048–1064, 2017. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/10494820.2016.1244084?casa_token=so4cyRTaJAwAAAAA%3AjANJO_p9FAR_D_oTZoovA9swJnofnpe5VEJYgUemjxx-QsvwK0kPwckv9t7r2vUcrBAIqnmRK_3Wig

[38] R. R. Choudhury, H. Yin, and A. Fox, "Scale-driven automatic hint generation for coding style," in *International Conference on Intelligent Tutoring Systems*. Springer, 2016, pp. 122–132.

[39] T. Barnes and J. Stamper, "Toward automatic hint generation for logic proof tutoring using historical student data," in *Proceedings of the 9th International Conference on Intelligent Tutoring Systems*. Citeseer, 2008, pp. 373–382. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.183.5576

[40] L. Fiorella and R. E. Mayer, "Eight Ways to Promote Generative Learning," *Educational Psychology Review*, vol. 28, no. 4, pp. 717–741, 4 2016. [Online]. Available: https://doi.org/10.1007/s10648-015-9348-9

[41] J. Paron and R. Mayer, "Learning Science in Immersive Virtual Reality," *Journal of Educational Psychology*, 2018.

[42] R. Garcia, K. Falkner, and R. Vivian, "Systematic literature review: Self-Regulated Learning strategies using e-learning tools for Computer Science," *Computers & Education*, vol. 123, pp. 150–163, 2018.

[43] Y. Semura, N. Yoshiday, E. Choi, and K. Inoue, "Multilingual Detection of Code Clones Using ANTLR Grammar Definitions," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 673–677. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8719568

[44] I. Cervesato, T. J. Cortina, F. Pfenning, and S. Razak, "An Approach to Teaching to Write Safe and Correct Imperative Programs—even in C." [Online]. Available: https://www.cs.cmu.edu/~./fp/papers/pic19.pdf

[45] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[46] T. Parr, S. Harwell, and K. Fisher, "Adaptive LL (*) parsing: the power of dynamic analysis," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 579–598, 2014.

[47] S. Medeiros and F. Mascarenhas, "Syntax error recovery in parsing expression grammars," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1195–1202.

[48] T. Dean and A. Fryer, "A Survey of Binary Protocol Parsers," *Available at SSRN 4246617*, p. 18, 2022. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4246617

[49] F. Ortin, J. Quiroga, O. Rodriguez-Prieto, and M. Garcia, "An empirical evaluation of Lex/Yacc and ANTLR parser generation tools," *Plos one*, vol. 17, no. 3, p. e0264326, 2022. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0264326

[50] R. Bodily, J. Kay, V. Aleven, I. Jivet, D. Davis, F. Xhakaj, and K. Verbert, "Open learner models and learning analytics dashboards: a systematic review," in *Proceedings of the 8th international conference on learning analytics and knowledge*. Proceedings of the 8th international conference on learning analytics and knowledge. 2018, 2018, pp. 41–50. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3170358.3170409?casa_token=u_BAIPZ2C0EAAAAA:pgPZXg1rYi3qQzoGYYq_ereLd3EvCnVft_-R9MYOHNVboT0e3l1xYxp6kKKg-z4RwoOXQzNu5fY

[51] M. G. Core, H. C. Lane, M. Van Lent, D. Gomboc, S. Solomon, and M. Rosenberg, "Building explainable artificial intelligence systems," in *AAAI*. 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference, 2006, pp. 1766–1773. [Online]. Available: https://markcore.github.io/papers/IAAI0602CoreM.pdf

[52] J. Hoffmann and D. Magazzeni, "Explainable AI Planning (XAIP): Overview and the Case of Contrastive Explanation," in *Reasoning Web. Explainable Artificial Intelligence*. Springer, 2019, pp. 277–282.

[53] M. Fox, D. Long, and D. Magazzeni, "Explainable Planning," *CoRR*, vol. abs/1709.1, 2017.

[54] J. C. Nesbit, Q. L. A. Liu, Q. Liu, and O. O. Adesope, "Work in progress: Intelligent tutoring systems in computer science and software engineering education," in *Proc. 122nd Am. Soc. Eng. Education Ann. Conf*, 2015, p. 12.

[55] J. Schneider, D. Börner, P. Van Rosmalen, and M. Specht, "Augmenting the senses: a review on sensor-based learning support," *Sensors*, vol. 15, no. 2, pp. 4097–4133, 2015. [Online]. Available: https://search-proquest-com.ezproxy.tees.ac.uk/docview/1663344230?pq-origsite=summon

[56] D. Dermeval, R. Paiva, I. I. Bittencourt, J. Vassileva, and D. Borges, "Authoring tools for designing intelligent tutoring systems: a systematic review of the literature," *International Journal of Artificial Intelligence in Education*, vol. 28, no. 3, pp. 336–384, 2018. [Online]. Available: https://link.springer.com/article/10.1007/s40593-017-0157-9

[57] A. T. Bimba, N. Idris, A. Al-Hunaiyyan, R. B. Mahmud, and N. L. B. M. Shuib, "Adaptive feedback in computer-based learning environments: a review," *Adaptive Behavior*, vol. 25, no. 5, pp. 217–234, 2017. [Online]. Available: https://journals-sagepub-com.ezproxy.tees.ac.uk/doi/full/10.1177/1059712317727590?utm_source=summon&utm_medium=discovery-provider#bibr10-1059712317727590

[58] K. Brawner, "Authoring Tools for Adaptive Training–An Overview of System Types and Taxonomy for Classification," in *International Conference on Augmented Cognition*. Springer, 2015, pp. 562–569. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-20816-9_54

[59] N.-T. Le and N. Pinkwart, "Towards a classification for programming exercises," in *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science*, 2014, pp. 51–60. [Online]. Available: https://publications.informatik.hu-berlin.de/archive/cses/publications/Towards-a-Classification-for-Programming-Exercises.pdf

[60] H. Ueno, "A generalized knowledge-based approach to comprehend pascal and C programs," *IEICE TRANSACTIONS on Information and Systems*, vol. 83, no. 4, pp. 591–598, 2000.

[61] I.-H. Hsiao, S. Sosnovsky, and P. Brusilovsky, "Adaptive navigation support for parameterized questions in object-oriented programming," in *Learning in the Synergy of Multiple Disciplines: 4th European Conference on Technology Enhanced Learning, EC-TEL 2009 Nice, France, September 29–October 2, 2009 Proceedings 4*. Springer, 2009, pp. 88–98. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1163405.1163411

[62] P. Brusilovsky and S. Sosnovsky, "Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, pp. 6–es, 2005.

[63] A. Núñez, J. Fernández, J. D. Garcia, L. Prada, and J. Carretero, "M-PLAT: Multi-programming language adaptive tutor," in *2008 Eighth IEEE International Conference on Advanced Learning Technologies*. IEEE, 2008, pp. 649–651. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4561793

[64] D. D. McCracken and E. D. Reilly, "Backus-naur form (bnf)," in *Encyclopedia of Computer Science*, 2003, pp. 129–131. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/1074100.1074155

[65] D. Quinlan, J. B. Wells, and F. Kamareddine, "BNF-Style Notation as It Is Actually Used," in *12th Conference on Intelligent Computer Mathematics 2019*. Springer, 2019, pp. 187–204. [Online]. Available: https://researchportal.hw.ac.uk/en/publications/bnf-style-notation-as-it-is-actually-used

[66] D. Crocker and P. Overell, "Augmented BNF for syntax specifications: ABNF," Tech. Rep., 2008. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc5234.txt.pdf

[67] M. Forsberg and A. Ranta, "The labelled bnf grammar formalism," *Department of Computing Science, Chalmers University of Technology and the University of Gothenburg*, 2005. [Online]. Available: https://bnfc.digitalgrammars.com/LBNF-report.pdf

[68] W. Zhu, N. Yoshida, T. Kamiya, E. Choi, and H. Takada, "MSCCD: Grammar Pluggable Clone Detection Based on ANTLR Parser Generation," *arXiv preprint arXiv:2204.01028*, 2022. [Online]. Available: https://arxiv.org/abs/2204.01028

[69] E. Wal, "Rosetta ANTLR: Ultimate Grammar Extractor," 2021. [Online]. Available: https://essay.utwente.nl/85728/

[70] J. Bovet and T. Parr, "ANTLRWorks: an ANTLR grammar development environment," *Software: Practice and Experience*, vol. 38, no. 12, pp. 1305–1332, 2008.

[71] R. Knöll and M. Mezini, "Pegasus: first steps toward a naturalistic programming language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 542–559. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1176617.1176628https://dl.acm.org/doi/10.1145/1176617.1176628

[72] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, "Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications," *Science of Computer Programming*, vol. 166, pp. 89–119, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642318301941?casa_token=bFZhowBc9z0AAAAA:HAJJOgcwSGMIoQGdw2xWNHZd7tLe_y7aAowC3EWomcj0HEMXfMjalqrTfS9YUa2uc538mXgB2e4

[73] OpenAI, "GPT-4 Technical Report," 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[74] P. P. Ray, "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266734522300024X

[75] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, and M. Gallé, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05100

[76] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. Günnemann, and E. Hüllermeier, "ChatGPT for good? On opportunities and challenges of large language models for education," *Learning and Individual Differences*, vol. 103, p. 102274, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1041608023000195

[77] A. Haleem, M. Javaid, and R. P. Singh, "An era of ChatGPT as a significant futuristic support tool: A study on features, abilities, and challenges," *BenchCouncil transactions on benchmarks, standards and evaluations*, vol. 2, no. 4, p. 100089, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772485923000066

[78] I. S. Chaudhry, S. A. M. Sarwary, G. A. El Refae, and H. Chabchoub, "Time to Revisit Existing Student's Performance Evaluation Approach in Higher Education Sector in a New Era of ChatGPT—A Case Study," *Cogent Education*, vol. 10, no. 1, p. 2210461, 2023. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/2331186X.2023.2210461

[79] N. M. S. Surameery and M. Y. Shakor, "Use chat gpt to solve programming bugs," *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, vol. 3, no. 01, pp. 17–22, 2023. [Online]. Available: http://journal.hmjournals.com/index.php/IJITC/article/view/1679

[80] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning Haskell," in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, 2003, pp. 62–71. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/871895.871902

[81] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000, pp. 268–279. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/351240.351266

[82] M. Lovett, O. Meyer, and C. Thille, "The Open Learning Initiative: Measuring the Effectiveness of the OLI Statistics Course in Accelerating Student Learning." *Journal of Interactive Media in Education*, 2008. [Online]. Available: https://eric.ed.gov/?id=ej840810

[83] M. Fox and D. Long, "PDDL2. 1: An extension to PDDL for expressing temporal planning domains," *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[84] M. T. H. Chi, N. De Leeuw, M.-H. Chiu, and C. LaVancher, "Eliciting self-explanations improves understanding," *Cognitive science*, vol. 18, no. 3, pp. 439–477, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0364021394900167

[85] O. Ladeinde and M. A. Razzaque, "Using Automated State Space Planning for Effective Management of Visual Information and Learner's Attention in Virtual Reality," in *Proceedings of SAI Intelligent Systems Conference*. Springer, 2019, pp. 24–40.

C# Example Code

C# was noted as the exception to obtaining code from the ANTLR4 GitHub repository to form tasks for the authoring tools. As there is no open source location for the C# "example" code used within this thesis, this section will be used to display the code used for the experiment, in the abscence of example code on the repository.

QuestionaireSetScript.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class QuestionaireSetScript : MonoBehaviour
{
    public EExpression TheStartImageIndex;
    public EExpression TheEndImageIndex;

    public string TheMainQuestion;
    public string[] TheQuestionaireQuestions;
    public string TheAnswerText
    {
        get
        {
            return AttachedConfirmedAnswer.text;
        }

        set
        {
            AttachedConfirmedAnswer.text = value;
```

```
            AttachedConfirmedAnswer.color = new Color(1f, 1f, 0.4f, 1f);
        }
    }

    private float my_confirm_bounce = 0f;
    private bool my_anim_active = false;
    private int my_answer_index = -1;
    public int TheAnswerIndex
    {
        get
        {
            return my_answer_index;
        }

        set
        {
            my_answer_index = value;
            TheAnswerText = (my_answer_index >= 0 && my_answer_index < TheQuestionair
                TheQuestionaireQuestions[value]:
                "Select an answer";

            my_confirm_bounce = 1f;
            my_anim_active = true;
        }
    }

    public QuestionaireButtonScript AttachedQuestionButton;
    public Sprite[] AttachedGUIExpressions;
    public Text AttachedQuestionText;
    public Text AttachedConfirmedAnswer;

    public enum EExpression
    {
        YAY = 0,
        FINE = 1,
        UNSATISFIED = 2,
```

```
    DIZZY = 3,
    SMILE = 4,
    MEH = 5
}

public void Generate_Questions()
{
    AttachedQuestionText.text = TheMainQuestion;

    for (int rep_question = 0; rep_question < TheQuestionaireQuestions.Length;
    {
        QuestionaireButtonScript quick_button = Instantiate(AttachedQuestionBut
        quick_button.transform.SetSiblingIndex(rep_question+2);
        if (rep_question == 0)
        {
            quick_button.TheSprite = AttachedGUIExpressions[(int)TheStartImageI
            quick_button.TheColour = QuestionaireButtonScript.EColour.LOW;
        }
        else if (rep_question == TheQuestionaireQuestions.Length -1)
        {
            quick_button.TheSprite = AttachedGUIExpressions[(int)TheEndImageInd
            quick_button.TheColour = QuestionaireButtonScript.EColour.HIGH;
        }
        else
            quick_button.TheColour = QuestionaireButtonScript.EColour.NONE;

        quick_button.TheIndex = rep_question;
        quick_button.TheText = TheQuestionaireQuestions[rep_question];

        quick_button.TheAttachedQuestionaire = this;
    }
}

// Start is called before the first frame update
void Start()
{
```

```
}

// Update is called once per frame
void Update()
{
    if (my_anim_active)
    {
        if (my_confirm_bounce <= 0f)
        {
            foreach (QuestionaireButtonScript rep_button in GetComponentsInChildr
            {
                //Squish animation
                rep_button.TheImage.transform.localScale = new Vector3(1f, 2f, 1f
            }
            AttachedConfirmedAnswer.transform.localScale = new Vector3(1f, 1f, 1f
            my_anim_active = false;
        }
        else
        {
            foreach (QuestionaireButtonScript rep_button in GetComponentsInChildr
            {
                //Squish animation
                rep_button.TheImage.transform.localScale = new Vector3(
                    1f / (1f + Mathf.Cos((my_confirm_bounce) * Mathf.PI * 2f) * (
                    2f * (1f + Mathf.Cos((my_confirm_bounce) * Mathf.PI * 2f) * (
                    1f);
            }
            AttachedConfirmedAnswer.transform.localScale = new Vector3(
                    1f / (1f + Mathf.Cos((my_confirm_bounce) * Mathf.PI * 2f) * (
                    1f * (1f + Mathf.Cos((my_confirm_bounce) * Mathf.PI * 2f) * (
                    1f);
        }
    }

    my_confirm_bounce -= Time.deltaTime * 5f;
```

```
    }

    public string Print_Answer()
    {
        return "Question: " + TheMainQuestion + "\nAnswer: " + (TheAnswerIndex+1) +
    }
}
```

QuestionaireGroupScript.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class QuestionaireGroupScript : MonoBehaviour
{
    private static QuestionaireGroupScript self;
    public static QuestionaireGroupScript TheActiveQuestionaire
    {
        get
        {
            return self;
        }
    }

    [Serializable]
    public struct QuestionaireForm
    {
        public string TheMainQuestion;
        public string[] TheQuestionaireQuestions;
        public QuestionaireSetScript.EExpression TheHighExpression;
        public QuestionaireSetScript.EExpression TheLowExpression;
    }

    public QuestionaireSetScript AttachedQuestionaire;
    public QuestionaireForm[] TheQuestions;

    public bool TheAutoGenerate = true;

    public void Generate_Questionaire()
    {
        foreach (QuestionaireForm rep_question in TheQuestions)
        {
            QuestionaireSetScript quick_question = Instantiate<QuestionaireSetScript>
```

```
            quick_question.TheStartImageIndex = rep_question.TheLowExpression;
            quick_question.TheEndImageIndex = rep_question.TheHighExpression;
            quick_question.TheMainQuestion = rep_question.TheMainQuestion;
            quick_question.TheQuestionaireQuestions = rep_question.TheQuestionaireQ

            quick_question.transform.SetParent(transform, false);
            quick_question.Generate_Questions();
        }
    }

    public int Check_Answer_Index(int its_question)
    {
        return GetComponentsInChildren<QuestionaireSetScript>()[its_question].TheAn
    }

    public bool Check_Answered()
    {
        foreach (QuestionaireSetScript rep_question in GetComponentsInChildren<Ques
        {
            if (rep_question.TheAnswerIndex < 0 || rep_question.TheAnswerIndex >= r
                return false;
        }
        return true;
    }

    public string Print_Answers()
    {
        string returner = "";
        int quick_index = 1;
        foreach (QuestionaireSetScript rep_question in GetComponentsInChildren<Ques
        {
            returner += quick_index++ + "-" + rep_question.Print_Answer();
        }
        return returner;
    }
```

```
    private void Awake()
    {
        self = this;
    }

    // Start is called before the first frame update
    void Start()
    {
        if (TheAutoGenerate)
            Generate_Questionaire();
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

GeneralNextButton.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class GeneralNextButton : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void On_Click_TaskComplete()
    {
        SurveyMaster.Load_Next_Task();
        GetComponent<Button>().interactable =  false;
    }
}
```

CSharpGrammar.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Antlr4.Runtime;
using Antlr4.Runtime.Tree;
using AntlrHarvest;
using Antlr4.Runtime.Misc;

public class CSharpGrammar : LanguageSettings
{
    public override IParseTree Create_Abstract_Syntax_Tree(string its_text)
    {
        AntlrInputStream quick_input = new AntlrInputStream(its_text);
        Lexer quick_lexer = new CSharpLexer(quick_input);
        CommonTokenStream quick_tokens = new CommonTokenStream(quick_lexer);
        CSharpParser quick_parser = new CSharpParser(quick_tokens);
        ParserRuleContext quick_ast = quick_parser.compilation_unit();
        Tree_Compare(its_text, quick_ast);
        return quick_ast;
    }

    public override string Language_Literal()
    {
        return "C#";
    }

}
```