Thesis

Strategy-based feedback for imperative programming exercises

Hieke Keuning
850412924
17 June 2014

Strategy-based feedback for imperative programming exercises

INSTITUTE	Open Universiteit The Netherlands Faculty of Management, Science and Technology Master Computer Science
STUDENT	Hieke Keuning (850412924)
SUPERVISOR	dr. Bastiaan Heeren
CHAIRMAN	prof. dr. Johan Jeuring
PRESENTATION DATE	17 June 2014

CONTENTS

Summary 1
1 Introduction 2
  1.1 Learning programming 2
  1.2 The Ideas project 5
  1.3 Problem statement and outline 5
2 Interactive domain-specific exercise assistants 7
  2.1 Domain reasoners 8
    2.1.1 Strategies 8
    2.1.2 Feedback 9
    2.1.3 Views 11
  2.2 Ask-Elle 11
    2.2.1 Refinement rules 12
    2.2.2 Strategies 12
    2.2.3 Normalisation 13
    2.2.4 Feedback services 13
    2.2.5 Results 14
3 Programming tutors 15
  3.1 Classification and definitions 15
    3.1.1 Focus 16
  3.2 Programming tutors before 2000 17
  3.3 Recent programming tutors 20
  3.4 Program assessment tools 27
  3.5 Conclusion 29
4 Research design 30
  4.1 Research questions 30
  4.2 Requirements and scope 30
  4.3 Validation 31
5 A domain reasoner for imperative programming 32
  5.1 Representation of imperative programs 33
  5.2 Strategies for imperative programming 37
    5.2.1 Rules 38
    5.2.2 Generating strategies 40
  5.3 Recognising solutions 46
    5.3.1 Similarity 46
    5.3.2 Equivalence 53
  5.4 Generation of strategy-based feedback 54

5.4.1 Diagnosis 54
5.4.2 Hints 58
5.4.3 Adapting feedback 59
6 Validation 63
6.1 Testing 63
6.2 Tutoring sessions 63
6.2.1 PHP tutoring session 63
6.2.2 Java tutoring session 66
6.3 Analysing student programs 69
6.3.1 PHP exercise analysis 69
6.3.2 Java exercise analysis 70
6.4 Conclusion 77
7 Conclusion 78
7.1 Answers to research questions 78
7.2 Contributions 79
7.3 Discussion and relation to similar work 80
7.4 Future work 80
8 Bibliography 83

SUMMARY

Ever since programming courses were introduced in schools and universities there has been active research into intelligent tutors that support students in learning programming. The recent emergence of large online courses and the limited availability of instructors increase the need for automated tools.

Designing a program is considered a difficult task. An important factor in learning is *feedback*, to inform a student how he or she is doing and where to go next. However, the actual programming process is often neglected in teaching methods. Our literature study on programming tutors shows that many tutors can only deal with finished programs and are not able to help the student in building the program step by step. Tutors are often limited to a single solution strategy for solving a programming problem and are difficult to adjust by instructors.

We have found that the existing *Ask-Elle* tutor for functional programming distinguishes itself by offering hints and feedback along the way towards a solution and is based on instructor-annotated model solutions. Ask-Elle uses the *Interactive domain-specific exercise assistants* (Ideas) framework to offer exercises and feedback services. The feedback the framework generates is based on *strategies*, which are sequences of steps for solving a problem.

In this thesis we report on our research into generating adaptable feedback to guide a student step by step towards a solution for an introductory programming problem that can be solved by multiple strategies for *imperative programming*. We have developed a prototype of a programming tutor using the Ideas framework.

We have designed an abstract syntax for simple imperative programs that includes a selection of basic imperative language constructs, such as loops, branching statements and variable assignments. We have developed a strategy generator that derives a programming strategy from a set of model solutions. The strategy describes the steps to arrive at one of these models. Steps can expand a program with new statements or gradually refine a particular expression. We incorporate alternative paths in the strategy for both the order of steps and some allowed variants of language constructs. To recognise even more variation, such as different variable names and expression forms, we perform a number of transformations on the program. We use the Ask-Elle feedback services for providing hints and diagnoses based on the strategy, while making some adjustments and additions to enable the services for a different programming paradigm. An evaluator has been created to inspect the output of a program. We have also implemented facilities for instructors to annotate a model solution to further control the feedback.

We demonstrate the capabilities of the prototype in a number of tutoring sessions in which is shown that our feedback leads the student to a correct solution. Student programs were collected during programming courses for first-year IT-students to establish to what extent we can recognise their solutions. We have found that we can recognise between 33% and 75% of the solutions that are similar to a model. Our suggestions for further research include expanding the programming strategy and improving feedback messages.

1 INTRODUCTION

Learning how to program is becoming increasingly important. Students learn programming in universities and colleges to become skilled software engineers. Many people, both young and old, teach themselves programming as a hobby or a new career path. Some high schools offer programming classes in their curricula.

Over the last few years many different initiatives have been advocating the importance of providing people with the opportunity to learn how to code. Computer programming education is being promoted actively. The underlying justification is both idealistic ('it teaches you how to think') but also stems from the shortage of software engineers currently and in the near future. Whatever the underlying reasons are, how easy is it for anyone to learn how to program and how can students be supported in their learning process?

This thesis describes our research on generating feedback for students doing programming exercises in a tutoring environment. In this chapter we look at the context of learning programming and the related difficulties. We also introduce the Ideas project that is of importance to our research. We formulate a problem statement and provide an outline of the remaining chapters in which we elaborate on our research.

"Everybody in this country should learn how to program a computer... because it teaches you how to think." - Steve Jobs

FIGURE 1 APPEALS TO START LEARNING PROGRAMMING

1.1 LEARNING PROGRAMMING

A small study examining the emotional state of students learning to program for the first time showed that after engaged (23%) the major emotions were confusion (22%), frustration (14%) and boredom (12%) (Bosch, Mello, & Mills, 2013). Dehnadi & Bornat report on the high failure rate of introductory programming courses (Dehnadi & Bornat, 2006), although Bennedsen and Caspersen (Bennedsen & Caspersen, 2007) are more careful about this claim. Another study (McCracken, Almstrum, & Diaz, 2001) shows that even after their first programming courses students do not know how to program. Therefore, students need all the help they can get to acquire the necessary skills to become successful in the field of computer science, a field that constantly asks for highly educated people.

Difficulties. In an international survey from 2005 (Lahtinen, Ala-Mutka, & Järvinen, 2005) on the difficulties of novices learning to program, in which over 500 students and teachers were questioned, the following issues were considered most difficult:

- Understanding how to design a program to solve a certain task.
- Dividing functionality into procedures.
- Finding bugs in one's own programs.

These findings are consistent with previous studies (Robins, Rountree, & Rountree, 2003; Soloway & Spohrer, 1989) that emphasise the importance of program design and applying the right programming constructs.

Teaching the actual programming *process* is considered important (Bennedsen & Caspersen, 2008). The programming process consists of a number of elements, among which the incremental development of a program by taking small steps and testing along the way. Another aspect is the refactoring of programs to improve their quality. Bennedsen and Caspersen note that traditional teaching methods such as textbooks and slide presentations do not cover this process. The authors propose process recordings as a way to teach the programming process but also state a long-term objective of programming education: *'… that students learn strategies, principles, and techniques to support the process of inventing suitable solution structures for a given programming problem.'*

At the same time learning has become more individual and is being done online more frequently. The traditional role of the teacher is changing. Teachers have limited time to spend on their students. In online courses, teachers do not even interact with their students in real life. A recent trend in education is the Massive Open Online Course, or MOOC.[1] These large-scale courses are often offered by renowned universities and can be done entirely through the internet. These developments depend heavily on digital tools to support the learning process.

Feedback. In the book 'Visible learning' by John Hattie (Hattie, 2008) more than 800 meta-studies on what is effective in education are analysed and summarised. *Feedback* has a very prominent position in the results. In a frequently cited article by Hattie and Timperley (Hattie & Timperley, 2007) the authors stress that the powerful influence of feedback on the learning process can either be positive or negative. A model is proposed to clarify how feedback can be put into practice in the best way. The findings are mainly about feedback from actual human beings, but because we want to closely mimic this in intelligent tutoring systems, several conclusions are of interest to our research. According to the model, the three questions that effective feedback should answer are:

- Where am I going? (feed up)
- How am I going? (feed back)
- Where to next? (feed forward)

These questions help learners to understand where they are right now and what has to be done to improve or finish a given task. The authors also claim that feedback is more effective when *'it builds on changes from previous trials'.* If these characteristics can be implemented in automated feedback systems, it will provide the student with a useful alternative to a human teacher.

Programming languages. There are many different programming languages and multiple programming paradigms. Imperative programming is defined as the stepwise execution of commands manipulating a program state. Its counterpart, declarative programming, is seen in the functional and logic programming paradigms. Functional programming revolves around the evaluation of mathematical functions, where the function is a first class value avoiding a program state. Another popular paradigm is object-oriented programming in which communicating objects, in which data fields and methods are encapsulated, are central.

There is no simple way to determine which programming paradigm is used the most or is most popular. A number of initiatives try to provide some insight into language popularity. In total eleven

---

[1] Pappano, L. (2012). The Year of the MOOC. The New York Times. Retrieved July 03, 2013, from http://www.nytimes.com/2012/11/04/education/edlife/massive-open-online-courses-are-multiplying-at-a-rapid-pace.html

languages appear in the top ten of the TIOBE Programming Community Index for April 2014[2] and the PYPL PopularitY of Programming Language index in May 2014[3] that were almost all there a year ago as well. All languages are based on the imperative paradigm, although most languages support other paradigms as well. It is therefore useful to focus our research on imperative programming.

Program variation. A major difficulty in creating automated tools for learning programming is the great diversity in possible solutions. As an example, a simple programming problem is given: calculate and print the sum of all odd positive numbers under 100. This problem can be solved in multiple ways using constructs of an imperative programming language, for example:

```
// option 1
int sum1 = 0;
for (int i = 1; i < 100; i = i + 2)
{
      sum1 = sum1 + i;
}
print(sum1);
```

```
// option 2
int sum2 = 0;
for (int i = 1; i < 100; i++)
{
      if (i % 2 == 1)
      {
            sum2 = sum2 + i;
      }
}
print(sum2);
```

```
// option 3
print(pow(100/2, 2));
```

We can also think of many variants for any of these solutions, for example:

```
// variant 1
int counter = 1;
int sum1 = 0;
while (counter <= 100)
{
      sum1 += counter;
      counter += 2;
}
print(sum1);
```

```
// variant 2
int x = 100 / 2;
int sum2 = x * x;
print(sum2);
```

---

[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, retrieved May 04, 2014

[3] https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language, retrieved May 04, 2013

```
// variant 3
print(2500);
```

In this small example we can already see many syntactic differences, such as:

- Using a while loop instead of a for loop.
- Using a compound assignment operator (counter += 2) instead writing out the full assignment (counter = counter + 2).
- Using a different name for a variable.

We can also identify a minor semantic difference in variant 1: looping until the counter is at least 101 instead of 100. The result is still a correct program. Also, if we swap two independent statements, do we get a different solution? Another issue is performing a calculation in steps instead of in a single assignment and even only printing the expected end result. Are these different solutions or simply variants of the same solution?

1.2 THE IDEAS PROJECT

In 2003 a group from the faculty of Computer Science of the Open Universiteit Nederland and the department of Information and Computing Sciences of Utrecht University started research on software technology for e-learning. This resulted in the Ideas project (Interactive domain-specific exercise assistants), in which *domain reasoners* are being developed that provide automated feedback while doing exercises in a learning environment.[4] Tutors have been developed for a number of different domains, such as mathematics and functional programming. The focus is on acquiring procedural skills such as solving equations and finding the solution to a programming problem. The Ideas project provides a framework that supports the stepwise solving of problems. These problems can often be solved with multiple solution strategies. Strategies can be specified using a strategy language to represent the various approaches to tackle the problem.

1.3 PROBLEM STATEMENT AND OUTLINE

Several important issues were highlighted in the previous sections, such as the difficulty of learning programming, the importance of feedback in the learning process and the absence of a human instructor, demanding an automated tutor that can adopt the task of providing feedback. We also know that students have a lot of difficulty with the design of programs and the actual programming process. In this thesis a research project is described that contributes to reducing this problem: we want to find out how to provide automated feedback for students solving an imperative programming problem. The Ideas project provides us with a software framework that serves as a basis for developing tutoring systems.

We provide an overview of the chapters of this thesis:

Chapter 2 Interactive domain-specific exercise assistants. We describe the Ideas framework that can be used to build tutors that help students solving exercises based on a solution strategy. We specifically focus on the tutor for functional programming that was developed using Ideas.

---

[4] http://ideas.cs.uu.nl/www, retrieved May 15, 2014

Chapter 3 Programming tutors. A literature study has been conducted on related work on tutors for programming. In this chapter we describe our findings and identify shortcomings in the field that we might respond to.

Chapter 4 Research design. In this chapter we propose our research based on the problem statement and conclusions from the literature research. We define several research questions, a corresponding method and validation questions.

Chapter 5 A domain reasoner for imperative programming. This chapter describes in detail the technical implementation of the tutor prototype. Knowledge of functional programming is required to fully understand the code fragments.

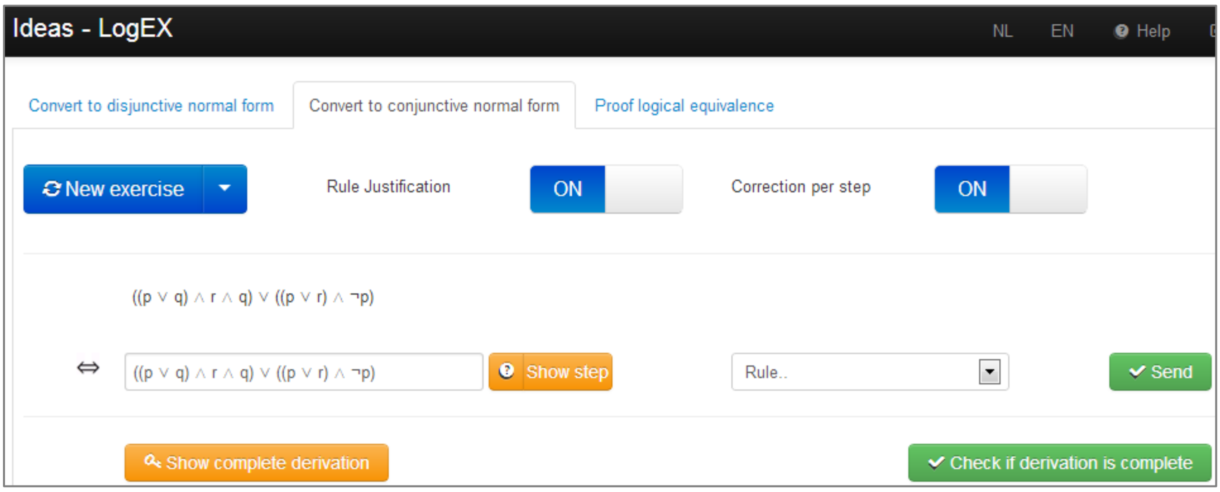
Chapter 6 Validation. We describe the capabilities of our prototype with respect to the validation questions by providing tutoring scenarios and analysing actual student data.

Chapter 7 Conclusion. We conclude with a chapter that summarises our contribution to the research on programming tutors. We also discuss the results, the outstanding issues and propose areas for further research.

2 INTERACTIVE DOMAIN-SPECIFIC EXERCISE ASSISTANTS

Interactive exercise assistants assist students in doing exercises on their own without the help of an instructor. The *Ideas framework* provides features to build these exercise assistants to help students to solve problems incrementally in various domains (Heeren & Jeuring, 2009b). Many skills, such as programming or mathematics, are acquired by learning strategies to solve exercises. These strategies help students to progress step by step and finally arrive at an approved solution. Students receive feedback on their progress and hints to move forward.

The Ideas framework is used for a number of different domains and their corresponding exercises: bringing a proposition in CNF (shown in Figure 2), orthogonalizing a set of vectors, relation algebra exercises and solving equations. To help students learn functional programming in Haskell, the tutor Ask-Elle has been created (Gerdes, Jeuring, & Heeren, 2012). The domains for which the Ideas framework is suitable are in general *well-structured* and their related problems can be solved in a procedural way. A well-structured domain has a formal and static body of knowledge that is only applicable in its own context, whereas *ill-structured domains* are more complex and variable with very specific contexts that are difficult to predict in advance (Spiro, Coulson, Feltovich, & Anderson, 1988).


Ideas - LogEX
NL
EN
Help
Convert to disjunctive normal form
Convert to conjunctive normal form
Proof logical equivalence
New exercise
Show step
Rule..
Send
Show complete derivation
Check if derivation is complete

Rule Justification ON

Correction per step ON

((p ∨ q) ∧ r ∧ q) ∨ ((p ∨ r) ∧ ¬p)

⇔ ((p ∨ q) ∧ r ∧ q) ∨ ((p ∨ r) ∧ ¬p)

FIGURE 2 A LOGIC TUTOR

The framework generates several different kinds of feedback (Heeren, Jeuring, & Gerdes, 2010):

- A check if the student has performed a valid step. The student can either be warned that he or she deviates from the proposed strategy or be forced to undo the last step.
- An indication of how close the student is to a solution, possibly by showing a progress bar or a minimum number of steps.
- A possible next step can be presented to the student by showing a hint or even providing the exact rule to apply.
- A check if the student has successfully completed the exercise.
- Showing the complete solution.

The levels of feedback the Ideas framework provides are based on the theory of VanLehn (Vanlehn, 2006) and Anderson (J. R. Anderson, 1993). Syntax errors or incorrectly applying some rule can also be detected by the system, but the main focus of the Ideas framework is to guide the student in selecting the right steps and progressing towards the solution.

Section 2.1 of this chapter describes domain reasoners, the software that provides feedback facilities based on strategies for a specific domain. The section is based on several publications on using strategies for exercise assistants (Heeren & Jeuring, 2009b), (Heeren et al., 2010), (Heeren & Jeuring, 2009a), (Gerdes, Heeren, & Jeuring, 2010). Section 2.2 focuses on the implementation of a functional programming tutor using the Ideas framework and is mainly based on two publications (Gerdes, Jeuring, et al., 2012; Jeuring, Gerdes, & Heeren, 2012).

2.1 DOMAIN REASONERS

To create an exercise assistant based on the Ideas framework, both a user interface and a *domain reasoner* should be created for a specific domain. A domain reasoner provides facilities to do exercises in the chosen domain and to generate personalised feedback and guidance. A domain reasoner can be built for a specific domain by implementing a number of components. The components should be built in the functional programming language Haskell using the Ideas software package.[5] The functionality of the domain reasoner is provided through stateless web services and can be used by any intelligent tutoring system.

2.1.1 STRATEGIES

The domain reasoners built on the framework are based on strategies. To specify strategies, we first need to establish the domain and a set of rules that apply to this domain. The *domain* can be described by a grammar for its abstract syntax and requires an accompanying parser to process submitted work. An example is the domain of propositional logic that comprises expressions consisting of variables and logic operators. An environment is maintained to store additional information, implemented as a set of key/value pairs.

Rules are transformations on the data type of the domain, such as refining or rewriting a student submission. An example in the logic domain is to rewrite the expression $p \rightarrow q$ as $\neg p \vee q$ using the rule of material implication. It is important to take into account the granularity of the rewrite steps. In some domains the student has to take each step explicitly, in other domains multiple simplifications can be done implicitly while solving the exercise step-by-step. Rules can be marked as either *major* or *minor*. Major rules are the main rules the student may apply and can be used as hints. Minor rules are generally more administrative.

Strategies are then used to describe the step-by-step solution to a problem. A strategy includes the possible steps, rewrite or refinement rules, together with the order in which the steps can be taken. For example, we can specify several strategies to solve the problem of simplifying a logical expression as much as possible. A strategy can be described using an *EDSL*, an embedded domain-specific language. The strategy description consists of a context-free grammar (CFG) and a part that is not context-free. The non-context-free part is implemented using a programming language and provides additional functionality.

Strategy *combinators* are used to compose more complex strategies. The atomic part in the CFG is a rule. An overview of combinators and operators for rules is given in Table 1. There are also a number of derived combinators based on the basic combinators, such as `many`, `many1`, `option`, `try`, `repeat` and $\triangleright$ (left-biased choice) and a set of traversal combinators.

---

[5] http://hackage.haskell.org/package/ideas, retrieved January 30, 2014

NAME	NOTATION	EXPLANATION
Sequence	`s <*> t`	Strategy `s` followed by strategy `t`.
Choice	`s <|> t`	Strategy `s` or strategy `t` can be applied.
Interleave	`s <%> t`	The steps of strategies `s` and `t` have to be applied, but can be interleaved, meaning the order of these steps is not relevant (Heeren & Jeuring, 2011). The interleave operator can also be used on sets of strategies. In this case all possible combinations of strategies from the left and the right set are allowed.
Atomicity	⟨s⟩	Marks the strategy `s` as atomic so interleaving is prohibited.
Label	`label l s`	Adds localised feedback messages to strategies.
Recursion	`fix f`	Returns the fixed-point of a function that maps a strategy to a new strategy.
Fail	δ	Always fails.
Succeed	ε	Always succeeds.
Applicability	`~s`	Specifies that the given strategy is not applicable to the current expression.

TABLE 1 STRATEGY COMBINATORS AND OPERATORS

2.1.2 FEEDBACK

To use strategies for the actual generation of feedback, an *exercise* should be specified. Exercises in the Ideas framework encompass all aspects related to a problem and the solving of that problem. The components of an exercise are:

- Meta data (identification code and description).
- A rule set, consisting of valid rules as well as *buggy rules* that identify common errors.
- A strategy to solve the problem.
- An equivalence relation and a similarity relation, possibly defined as *views* (see Section 2.1.3). The similarity relation is more tolerant than equivalence and used to detect correct but insignificant steps.
- Predicates to check if a starting expression is suitable (pre-condition) and to detect solved expressions (post-condition).
- A parser that detects syntax errors and, if necessary, a pretty-printer.
- Optionally, a randomised expression generator.
- A rule ordering function.

The domain reasoner consists of a set of exercises and a number of services to support the student doing the exercises. We summarise some of the feedback services in Table 2.

SERVICE	EXPLANATION
EMPTY	Check if the language of a strategy contains the empty sentence or a sentence consisting of minor rules.
FIRSTS	Splits a strategy into its first rule or applicability check together with the remaining strategy.
STEP	The application of a rewrite rule.
RUN	Applying the rules specified in a strategy to a term. The result is a (list of) solution(s) to the problem.
TRACE	An extension of run that shows the intermediate steps.
DIAGNOSE	Diagnoses an expression submitted by a student.

TABLE 2 STRATEGY SERVICES

The language that can be generated by the strategy grammar is a set of sequences of rules. These sentences can be compared with (partial) student solutions to see if they are on the right track. The student solution should be a prefix of a sentence or a complete sentence. The Ideas framework uses its own strategy recogniser because existing parsers are not entirely suited to the problem of recognising the application of rewrite rules, the detection of errors therein and the ability to provide informative feedback. Rule ordering is used to make a choice between multiple applicable rules so the provided feedback is deterministic.

In Table 3 the different types of diagnoses returned by the DIAGNOSE service are described. The input to this service is the submitted term and the previous state. A *state* is the product of an environment, an expression in focus (a zipper) and a strategy and is used to capture (intermediate) results. Apart from the type of the diagnosis, additional information is returned such as a possible new state, the rule that was applied and a boolean indicating if the student has finished the exercise. There is no new state when the diagnosis is either ‘buggy’ or ‘not equivalent’.

DIAGNOSIS	EXPLANATION
Buggy	The submitted term is not equivalent to the previous term because a buggy rule was applied.
Not equivalent	The submitted term is not equivalent to the previous term through an unknown mistake.
Wrong rule	A chosen rule is applied incorrectly.
Expected	The submitted term is expected by the strategy.
Similar	The submitted term is very similar to the previous submission.
Detour	A rule is (correctly) applied but does not follow the strategy.
Correct	Although the submitted term is equivalent to the previous, an unknown step is taken.
Unknown	Not used.

TABLE 3 IDEAS DIAGNOSE TYPES

2.1.3 VIEWS

Comparing the *canonical forms* of two expressions can determine if they are equivalent. A canonical form, or normal form, is a standard way of presenting an expression. A *view* defines a canonical form and consists of two functions:

- A *match* function that attempts to map an expression to a canonical form. An example is to return an expression with a plus at top-level. As an example, 1 – 2 will be transformed into 1 + -2. A match function usually converts an expression into a different type, such as a tuple of operands (1, -2) for the given example.
- A *build* function that maps a canonical form to an expression. A corresponding build function for the previous example might be transforming the tuple (1, -2) into the expression 1 - 2.

Performing a match operation followed by a build operation results in the canonical form of an expression. Views can also be composed of other views by using *arrow-combinators* (Paterson, 2003) such as `>>>` for a sequence of views and `***` for a parallel composition of views.

Besides equivalence checking, views are used in several ways:

- As a rewrite rule.
- To check if a term has a canonical form.
- To limit the set of necessary rewrite rules because the number of cases that occur greatly decreases.

2.2 ASK-ELLE

The interactive tutor Ask-Elle supports students in doing simple exercises to learn the functional programming language Haskell. The tutor is targeted at students in their first year of computer science. The tutor has the following features:

- Providing hints at each step.
- Providing feedback on the progress.
- Providing solutions.
- Recognizing common errors.

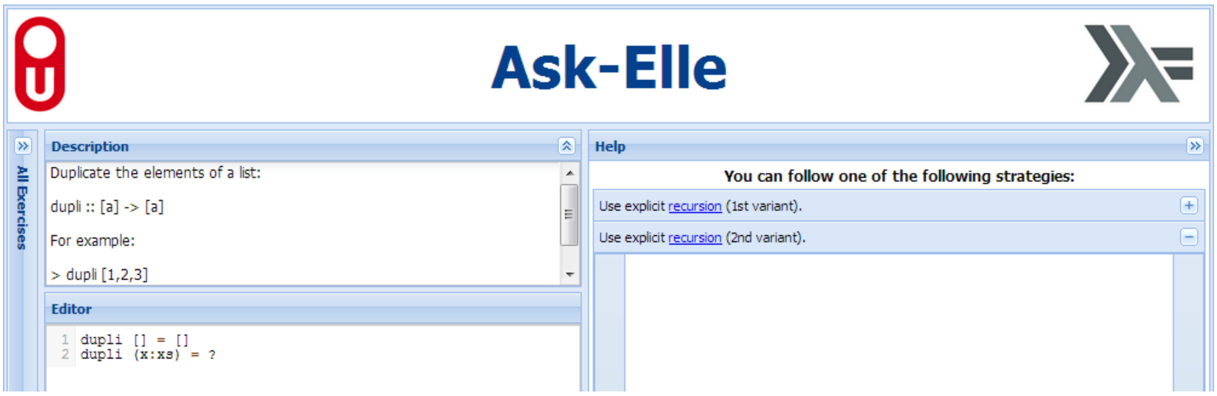
The tutor can be accessed online through a web application[6], as shown in Figure 3. The tutor is easy to use by students and easy to customise by instructors. The student can select an exercise and solve it step-by-step. Unfinished parts of the solution can be indicated by a ●-sign. Instructors can add their own programming exercises to the tutor, together with a set of model solutions.

The Ask-Elle tutor distinguishes itself from other tutors with the following characteristics:

- The tutor handles incomplete programs.
- Performing multiple steps at once is recognised by the tutor.
- Feedback is calculated automatically and is based on model solutions.
- Correctness is determined by equivalence to a model solution.

---

[6] http://ideas.cs.uu.nl/ProgTutor, retrieved September 02, 2013



Duplicate the elements of a list:

dupli :: [a] -> [a]

For example:

> dupli [1,2,3]

```
dupli [] = []
dupli (x:xs) = ?
```

You can follow one of the following strategies:

Use explicit recursion (1st variant).

Use explicit recursion (2nd variant).

FIGURE 3 A SCREENSHOT OF ASK-ELLE IN ACTION

A domain reasoner for the functional programming domain has been implemented for this tutor. A Haskell compiler is used to provide syntax error and type error messages. In this section the components of the domain reasoner are described together with the deviations and extensions to the existing Ideas framework.

2.2.1 REFINEMENT RULES

The rules used in the functional programming domain are *refinement rules*, instead of the rewrite rules that are used for mathematical domains. Refinement rules are used to make a program more complete instead of transforming a program into a semantic equivalent when applying rewrite rules. By applying refinement rules, holes in an expression can be replaced by new expressions. Refinement rules refine expressions, declarations, function bindings, alternatives and patterns. Examples of rules are introducing a variable or introducing an if-then-else construct. These new expressions can also contain holes, for example `if ● then ● else ●`. Refinement rules are atomic and cannot be divided into smaller rules. The available rules cover all available language constructs in the abstract syntax. Rules can be defined as *minor* if they should be performed together with another refinement rule. Rules are always applied at a specific hole location in the student program. Holes are therefore tagged with a unique identifier so rules can be matched with a location identifier.

2.2.2 STRATEGIES

The tutor considers the step-by-step, top-down application of refinement rules as the strategy to solve functional programming exercises. Strategies can be used to secure the order in which rules are applied. For example, some refinement rules can only be performed after another refinement rule. The strategy for solving a programming problem is automatically derived from the set of model solutions provided by the instructor. The strategy combines the multiple strategies from different solutions into one strategy. This strategy is used to provide feedback. A strategy can be quite strict, which could be a disadvantage. On the other hand, the student is guided towards a solution that is preferred by an instructor.

Both partial and complete programs are converted into a canonical form so equivalent variants can be taken into account. A program is correct if it follows a strategy. Two programs are similar if their canonical forms match syntactically. If a program does not follow a known strategy, the program is considered equivalent to the solution if it passes a number of tests. A modified version of the QuickCheck library (Claessen & Hughes, 2000) that is able to handle partial programs is used to check a number of properties about the program. This is not entirely reliable because checking program equivalence is undecidable and the testing might not identify all faults in the program.

The instructor can specify feedback messages by annotating the model solutions. Information regarding the exercise as a whole can be provided as meta-data in a configuration file. Textual feedback is specified in feedback scripts that are automatically generated and can be adjusted by the instructor. Refinement rules and library functions have an accompanying feedback message.

Feedback can be adapted by the instructor in a number of ways:

- Providing a general description of the model solution.
- Specifying alternative code to a prelude (the standard Haskell library) function, so more possible solutions will be accepted by the tutor.
- Enforcing the use of specific constructs in the solution.
- Adding localised feedback messages.

A strategy is derived by matching the language constructs in the AST with the available refinement rules. The interleave (`<%>`) combinator is used to allow variation in the order of refining certain expressions. Feedback annotations, such as an ALTERNATIVE, are included at the right locations. If library functions are encountered, the corresponding definition will also be included in the strategy, except when a MUSTUSE-annotation is provided by the instructor.

As an example we show an annotated model solution for reversing a list of elements:

```
{-# DESC Use the prelude function foldl. #-}
    reverse =
    {-# FEEDBACK foldl takes an operator and a base value as argument. #-}
        (foldl {-# FEEDBACK Use flip and (:). #-} (flip (:)) [])
```

The strategy derived from this model solution is (omitting feedback labels):

```
    patBind
<*> pVar "reverse"
<*> app
<*> var "foldl"
<*> (    (paren <*> app <*> var "flip" <*> infixApp <*> con "(:)" )
      <%> con "[]"
    )
```

2.2.3 NORMALISATION

To recognise programs that are equivalent but syntactically different, programs are converted to a canonical form. The program transformations are independent from the strategy for the exercise. The following transformations, based on the lambda-calculus, are performed:

- Inlining: replacing a call to a user-defined function by its body, dead-code elimination and constant argument removal.
- Desugaring: removing syntactic sugar from programs.
- α-renaming: assigning a new name to all variables.
- β-reduction: function application.
- η-reduction: replacing a function with the form `\x -> f x` by `f`.

2.2.4 FEEDBACK SERVICES

In Table 4 two additional strategy services are described that were added for the Ask-Elle tutor.

SERVICE	EXPLANATION
DEEPDIAGNOSE	Recognises multiple steps instead of just a single next step.
TASKDESCRIPTION	Returns feedback messages for all active labels.

TABLE 4 ADDITIONAL SERVICES

Each time a student asks for a hint, the partial program the student was working on will be submitted and normalised. All refinement rules that follow the exercise strategy are applied to the previous state. This may result in multiple programs that will all be normalised and compared to the normalised student program. For the DEEPDIAGNOSE service a top-down parallel recogniser is used that is able to work with programs that have the same first step (a left factor in the generated strategy). In the state, information is added on which strategies are still possible, instead of having to choose one when the student has only completed the overlapping part.

A challenge for the strategy recogniser is to recognise multiple steps at once. If a program contains many holes that may be filled in any order, the number of correct next steps greatly increases. The AST is searched depth-first attempting to reduce search time based on the location where the student most likely proceeds. The tree can be further pruned by reducing the number of interleave options because the order is irrelevant when recognising multiple steps.

Strategies can also be used for assessing complete Haskell programs so the instructor does not have to grade them by hand (Gerdes, Jeuring, & Heeren, 2010). This is a different approach from most assessment tools that use testing to check student solutions. The advantage of using strategies is that the design is assessed and not only the outcome. It can also be proven that a student solution is equivalent to a model solution written by an instructor, whereas proving that a program is correct based on test results is in general impossible.

2.2.5 RESULTS

In 2011 an experiment with around 100 students using Ask-Elle was performed. The emphasis of this experiment was on the findings of the students, who were moderately enthusiastic. The main focus of the criticism was that not all correct solutions were recognised and that proof for an incorrect solution was absent. In an experiment with Ask-Elle as an assessment tool, 89% of the correct programs were recognised and no incorrect programs were marked as correct (Gerdes, Jeuring, et al., 2010). A total of over 90 programs were assessed with only four model solutions.

Several options for further research have been proposed (Gerdes, 2012). Some are focused on the area of usability and the didactic results. Technological improvements focus on supporting the full Haskell standard and adding new program transformations. Research on new functionality includes offering other types of exercises, supporting larger programs, providing better error information and developing a tutor for other programming languages and paradigms.

3 PROGRAMMING TUTORS

Ever since learning programming was introduced in schools and universities, many tools have been developed to support students in learning how to program. A literature study has been conducted to provide an overview of the evolution of programming tutors that automatically provide feedback to its users. The characteristics and features of the tutors are identified and compared to those of the Ideas tutors. We describe the classification of tools and problems in Section 3.1, together with some relevant definitions. In Section 3.2 some classic and influential tutors from the 20th century are examined. Section 3.3 focuses on modern tutors. In Section 3.4 we take a closer look at program assessment tools. Assessment tools are related to the diagnosis of student programs and are therefore of interest to the development of programming tutors. We conclude in Section 3.5 by summarising the most striking observations.

3.1 CLASSIFICATION AND DEFINITIONS

Tools and environments that help students learning to program come in many forms. There are a large number of different classifications of programming tools (Deek & McHugh, 1998; Gomez-Albarran, 2005; Guzdial, 2004; Pausch & Kelleher, 2005). In 1998 Deek and McHugh (Deek & McHugh, 1998) recognised *intelligent tutoring systems* (ITS) as well as *intelligent programming environments* as two of the four major groups. In 2005 Gómez-Albarrán (Gomez-Albarran, 2005) proposed a different classification. The author did not include ITSs as a distinct category because of the observation that newer tools focus less on providing intelligent tutoring. Intelligent systems reappear in a survey of literature on the teaching of introductory programming from 2007 (Pears, Seidman, Malmi, & Mannila, 2007), in which the following categories for tools that support teaching programming are used:

- Visualization tools.
- Automated assessment tools.
- Programming environments for novices, divided into programming support tools and Microworlds.
- Other, including intelligent tutoring systems.

An *intelligent tutoring system* is a learning system that provides automated feedback or instruction to the user of the system (Nwana, 1990). The interaction with the system should be comparable to interaction with an actual teacher. Research on ITSs involves the domains of computer science, cognitive psychology and education. ITSs generally consist of four different components: the expert knowledge module (domain), the student model module, the tutoring module and the user interface.

To correctly analyse student programs, the following functionality should be provided by an ITS (Vanneste, 1994):

- Handle variation in syntax.
- Recognise different algorithms.
- Recognise errors in implementations.
- Analyse program efficiency.
- Acknowledge if the system is unable to deal with a program, and direct the student to a teacher.
- Generate feedback the student can understand.

Pillay (Pillay, 2003) distinguishes two major functions in ITSs. The first group of tutors helps a student in writing a program for a certain problem. The second group of tutors focuses more on debugging and error diagnosis. A commonly used term is *intention-based diagnosis* (L. Johnson, 1986), which identifies errors through attempting to understand what the student was trying to achieve and how he or she wanted to achieve that, instead of just reporting on what is wrong.

In addition to the classification of tools, a classification based on the nature of a problem can also be made. Le et al. (N.-T. Le, Loll, & Pinkwart, 2013) propose a classification of educational problems based on their degree of *ill-definedness*. This classification is based on three properties of solution spaces: alternative solution strategies, implementation variability and solution verifiability. In programming, there are both well-defined and ill-defined problems. Moreover, programming problems in each category can be devised, as shown in Table 5.

CLASS	DEFINITION BY LE ET AL.	EXAMPLE PROGRAMMING PROBLEM
1	One solution strategy, one implementation.	Choose the right operators to calculate degrees Celsius to Fahrenheit: `int f = c ? 1.8 ? 32;`
2	One solution strategy, alternative implementation variants.	Write a *recursive* function that calculates the factorial of a number n.
3	A known number of typical solution strategies.	Write a function that calculates the factorial of a number n.
4	A great variability of possible solution strategies while the correctness of any given specific solution can be verified automatically.	Write an application that can convert between multiple temperature scales.
5	Multiple solution strategies, and solution correctness cannot be verified automatically.	Write a platform computer game.

TABLE 5 PROBLEM CLASSIFICATION WITH EXAMPLES

Class 1 problems are usually addressed by CAI-systems (computer aided instruction) that only have to mark the student's solution as correct or incorrect. *Model-tracing* and *constraint-based* modelling techniques are usually applied for class 2 problems. Model-tracing focuses on the process and uses rules to model the steps a student may take to solve a problem (Kodaganallur, Weitz, & Rosenthal, 2005). Constraint-based modelling, on the other hand, focuses on the product: it models constraints as conditions that must be met in the end result. Tutors based on constraint-based modelling are not interested in *how* to arrive at the end result. Class 3 problems are strategy-based. Systems for class 4 are rare and mainly use data-mining techniques. Class 5 problems lean on heuristic techniques. The authors observe that solutions for class 3 and 4 are the least developed although they are of great importance for learning and developing problem-solving skills.

The main objective for this classification is twofold: it provides guidance for developers of educational systems but it also proposes a common language to specify problems in order to gain more insight into the capabilities of an educational system. The Ask-Elle programming tutor can be categorised in class 3.

3.1.1 FOCUS

Considering the classification from Section 3.1, the following categories are of interest for this research on programming tutors:

- Automated assessment tools.
- Intelligent tutoring systems.

The tutors based on the Ideas framework, and in particular the functional programming tutor, distinguish themselves by the following characteristics (Gerdes, Jeuring, et al., 2012):

- Dealing with incomplete programs.
- The recognition of multiple steps in a student program.
- Automated calculation of feedback based on model solutions.
- Correctness determined by equivalence to a model solution.

We examine several programming tutors considering the features mentioned above, the criteria by (Vanneste, 1994) and the classification by Le et al (N.-T. Le et al., 2013). For this research we are mostly interested in the knowledge model and the tutoring module of an ITS, and not the student module and the user interface. The tutors and tools should be able to deal with class 2 or class 3 programming problems.

3.2 PROGRAMMING TUTORS BEFORE 2000

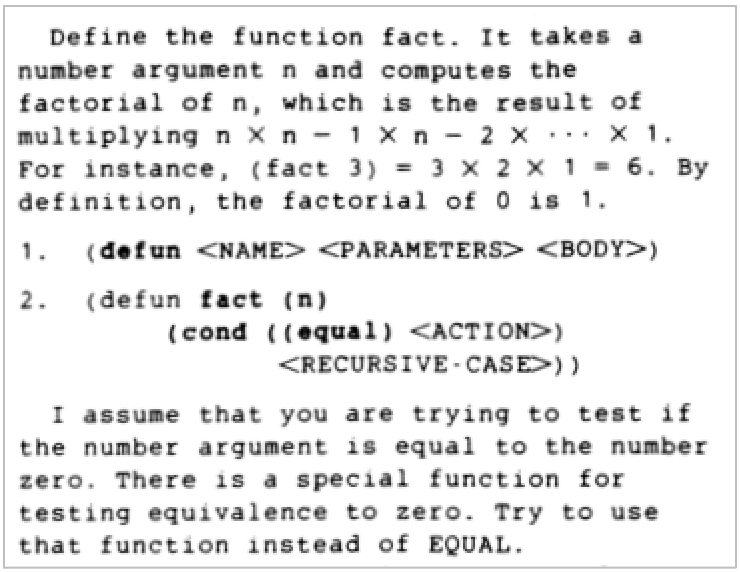
The tutors described in this section have been selected based on five publications that categorise and review learning tools for programming (Deek & McHugh, 1998; Gomez-Albarran, 2005; Guzdial, 2004; Pausch & Kelleher, 2005; Pillay, 2003), the frequency a certain tool appears in these publications and the number of citations. Table 6 provides an overview of the features of the tutors that are discussed.

	THE LISP TUTOR	PROUST	C-TUTOR	CM STRUCTURE EDITORS
Appearance in review articles	4/5	3/5	0/5	2/5
Google Scholar citations	179	383	49	57 (many more to related articles)
Main paradigm	Functional	Imperative	Imperative	Imperative
Problem classification	Class 1 and 2	Class 3	Class 3	Class 2
Feedback and hints	✓	✓	✓	✗
Handles partial programs	✓ , but only from left to right	✗	✗	✓
Handles multiple steps	✓	-	-	✓
Knowledge base	Problem and production rules	Problem, goals, plans, rules	Model program, test data	Assignment templates
Handling of syntax variation	Unknown	✓ , but limited according to (Song, Hahn, Tak, & Kim, 1997)	✓	✓
Handling of algorithm variation	-	Programming plans	Goals and plans	-
How to assess program correctness	Model solution	Plan extracted from model solution	Plan extracted from model solution	Model solution

TABLE 6 FEATURES OF PROGRAMMING TUTORS

The Lisp Tutor. The Lisp Tutor was developed in the 1980s at Carnegie-Mellon University using techniques from artificial intelligence and cognitive psychology (J. Anderson & Skwarecki, 1986). By observing how students program, the developers created simulation models that can be used to guide the student during programming. These models were implemented using *production systems* that set goals and subgoals. The system uses model tracing to keep track of the students' actions.

A student working with the tutor can solve a problem step-by-step by filling in a coding template and answering questions about the problem. The possible next steps students can take can be calculated from the rules in the production system. The production system also contains buggy rules so the tutor can respond to students doing something wrong.

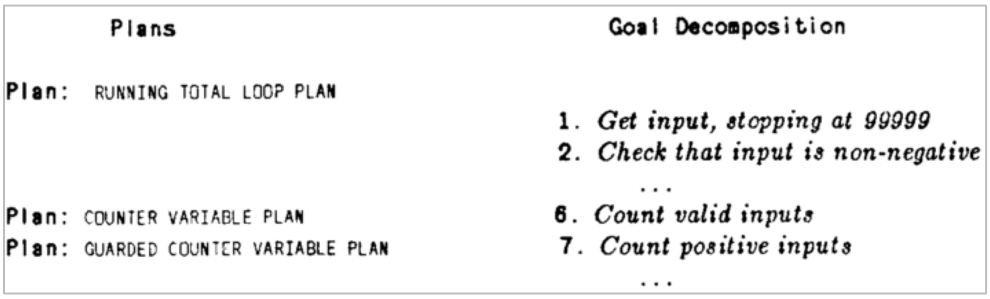


```
  Define the function fact. It takes a
number argument n and computes the
factorial of n, which is the result of
multiplying n × n − 1 × n − 2 × ··· × 1.
For instance, (fact 3) = 3 × 2 × 1 = 6. By
definition, the factorial of 0 is 1.

1.   (defun <NAME> <PARAMETERS> <BODY>)

2.   (defun fact (n)
          (cond ((equal) <ACTION>)
                <RECURSIVE-CASE>))

  I assume that you are trying to test if
the number argument is equal to the number
zero. There is a special function for
testing equivalence to zero. Try to use
that function instead of EQUAL.
```

FIGURE 4 A FRAGMENT OF THE LISP TUTOR IN ACTION

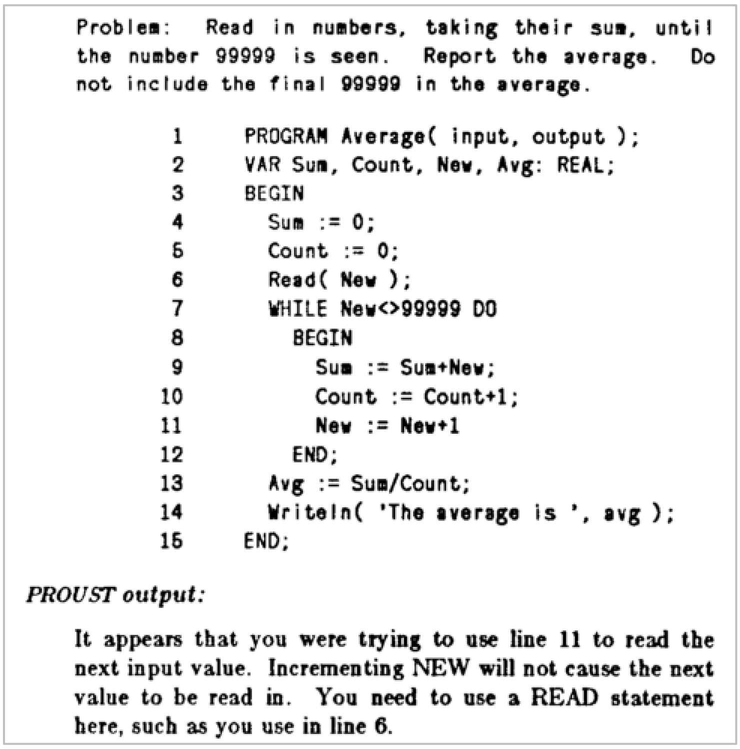
The approach was successful because using the tutor led to better results than learning to program Lisp without it. Nevertheless a number of improvements were discussed for future development. A known issue is the top-down, left-to-right programming style that is enforced which causes the tutor to be inflexible and restrictive. The interaction is also limited and adding new exercises is laborious.

PROUST. PROUST is a programming tutor for learning Pascal (W. Johnson & Soloway, 1985). The tutor analyses complete programs that solve a certain problem and returns an explanation of the discovered bugs using *programming plans*. A programming plan is '*a procedure or strategy for realising intentions in code'* and is decomposed into a number of *goals,* as illustrated in Figure 5. One programming problem may have different goal decompositions. PROUST tries to recognise these plans, including erroneous plans, in the code submitted. Transformation rules are used to detect variations in implementation.



Plans	Goal Decomposition
Plan: RUNNING TOTAL LOOP PLAN	
	1. *Get input, stopping at 99999*
	2. *Check that input is non-negative*
	. . .
Plan: COUNTER VARIABLE PLAN	6. *Count valid inputs*
Plan: GUARDED COUNTER VARIABLE PLAN	7. *Count positive inputs*
	. . .

FIGURE 5 PLANS AND GOALS USED BY PROUST



Problem: Read in numbers, taking their sum, until the number 99999 is seen. Report the average. Do not include the final 99999 in the average.

```
 1     PROGRAM Average( input, output );
 2     VAR Sum, Count, New, Avg: REAL;
 3     BEGIN
 4       Sum := 0;
 5       Count := 0;
 6       Read( New );
 7       WHILE New<>99999 DO
 8         BEGIN
 9           Sum := Sum+New;
10           Count := Count+1;
11           New := New+1
12         END;
13       Avg := Sum/Count;
14       Writeln( 'The average is ', avg );
15     END;
```

PROUST output:

It appears that you were trying to use line 11 to read the next input value. Incrementing NEW will not cause the next value to be read in. You need to use a READ statement here, such as you use in line 6.

FIGURE 6 RESPONSE BY PROUST TO A BUGGY PROGRAM

Looking at the results, the number of correctly analysed programs seems to be satisfactory (72% of 206 student solutions were completely analysed and 22% partially), although in (Gomez-Albarran, 2005) it is noted that the system has problems with analysing programs with increased complexity. Adding a new exercise and the corresponding goals, plans and rules requires substantial work.

C-tutor. The C-Tutor is an ITS for beginners learning programming in C (Song et al., 1997). The system consists of a learning environment and a program analyser and is based on intention-based diagnosis. The program analyser includes a reverse engineering system and a didactic system. The reverse engineering system generates the problem description, the *intention*, based on a model program provided by an instructor. The problem description contains a hierarchy of programming goals and plans. The student submits a solution, which is then converted into a canonical form. The didactic system generates feedback on the solution using both dynamic and static analysis. The system runs test cases provided by the instructor, tries to match the student program to the plans extracted from the model solution (similar to the approach in PROUST) and reports the bugs to the student.

The C-tutor is successful in the recognition and analysis of student programs: 93% of 240 programs were successfully analysed. Using both static and dynamic analysis is a strong point of the tutor, besides the easy addition of new exercises. However, the tutor is unable to guide a student through the programming process step-by-step.

Carnegie Mellon Structure Editors. Carnegie Mellon University developed three generations of novice programming environments, GNOME, MacGnome (Genies) and ACSE for Pascal and other languages, starting from the 1980s (Miller, Pane, Meter, & Vorthmann, 1994). The development focused on educational concepts such as procedural abstraction, data abstraction and reasoning about programs. The programming environments are *structure editors*, in which the student builds up an abstract syntax tree while creating a program. Parts of the tree that have not been realised yet are indicated by a textual placeholder. In this way all constructed programs are syntactically correct because the tree can only be elaborated with legal operations.

The tools were successful and widely used for many years, although there was some criticism and room for improvement. The technology was unable to adapt itself quickly to new languages. Criticism came from computer scientists who were bothered by the lack of focus on program correctness. The tools do not really provide feedback on the progress of a student, neither are they able to help a student to solve a programming problem in multiple ways.

3.3 RECENT PROGRAMMING TUTORS

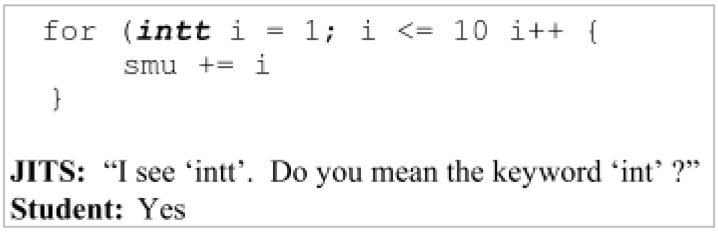
In this section a number of interesting and relevant tutors are discussed that were developed more recently. Table 7 provides an overview of their features.

	JITS	J-LATTE	PROLOG TUTOR	PYTHON TUTOR	DATA-DRIVEN PYTHON TUTOR
Main paradigm	Imperative	Imperative	Logic	Imperative	Imperative
Problem classification	Class 3	Class 2	Class 3	Class 2	Class 3
Feedback and hints	✓	✓	✓	✓	✓ , in progress
Handles partial programs	✗	✓	✓ , only on a high level	✗	✓ , but with large deltas (save/compile points)
Handles multiple steps	-	✓	✓	-	✓
Knowledge base	A problem, the required output and a program skeleton	Formal problem specification	Model solutions	Model solution and error model	Student data
Handling of syntax variation	✓	✓	✓	✓	✓
Handling of algorithm variation	✓	✗	✓	✗	✓
How to assess program correctness	Output	Constraints	Model solution	Model solution	Student solutions

TABLE 7 FEATURES OF PROGRAMMING TUTORS

JITS. Around 2003 a prototype of the Java Intelligent Tutoring System (JITS) was designed for students in their first programming course at college or university level (Sykes & Franek, 2003). The tutor focuses on a subset of the Java programming language (variables, operators and looping structures) and incorporates findings from artificial intelligence and cognitive science. Students can practise with programming exercises and ask for hints. The prototype version of JITS provides two types of functionality:

- *A-type* functionality. When a programming exercise has a straightforward solution, the system calculates a transformation string from the student's submission to the solution provided by the lecturer. A transformation string contains symbols that indicate which changes have to be made to transform the source into the target string. The system tries to implement the changes from the transformation string in a stepwise fashion through interaction with the student.
- *B-type* functionality. In this case there are multiple solutions and the lecturer only provides the problem description and the desired results. An intent recognition scanner-parser algorithm is used to determine the intention of the student. The student's code is analysed step-by-step and applies transformation sequences to produce tokens the parser can recognise, using a symbol table and a list of reserved words and keywords.



```
for (intt i = 1; i <= 10 i++ {
     smu += i
}
```

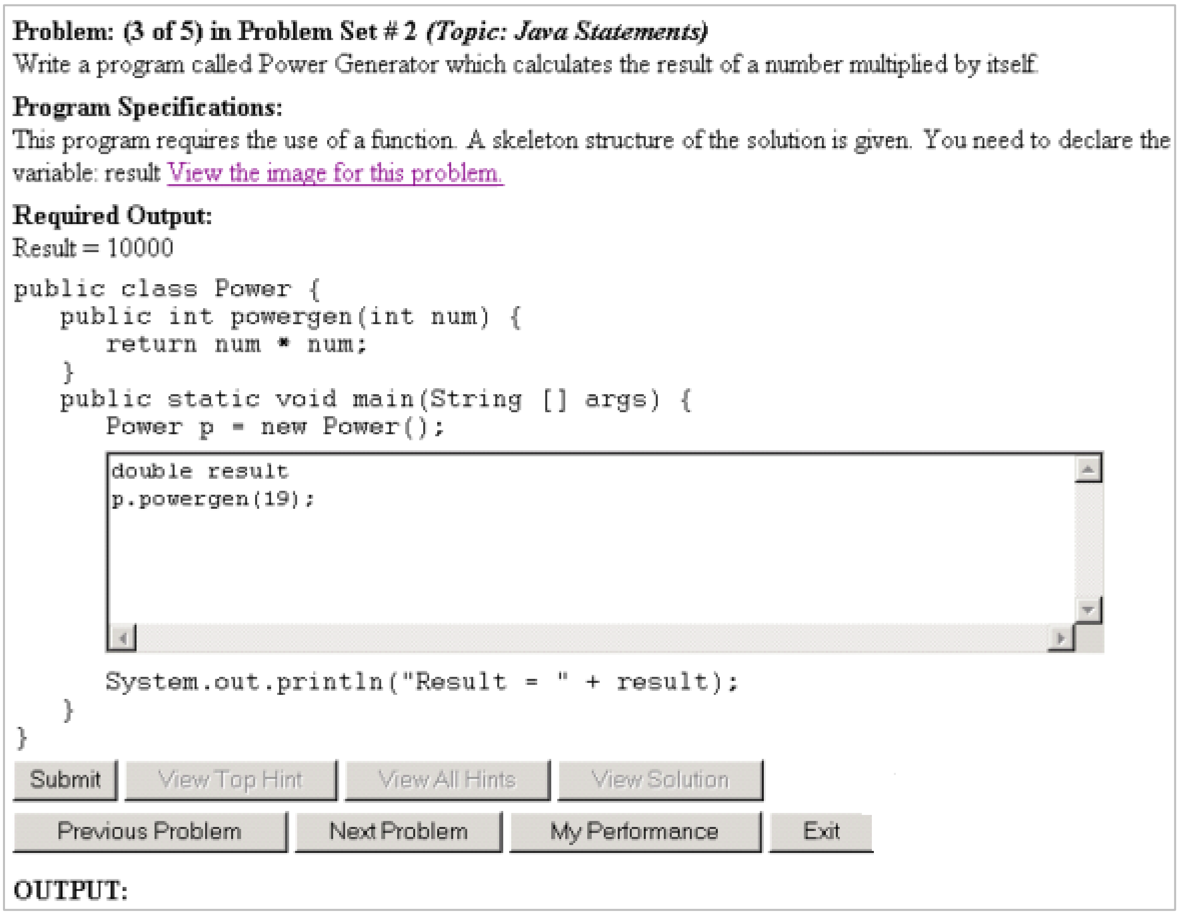
JITS: "I see 'intt'. Do you mean the keyword 'int' ?"
Student: Yes

FIGURE 7 THE TUTORING PROCESS IN THE JITS PROTOTYPE

JITS continued to be developed and tested resulting into an updated version (Sykes, 2005). In this version instructors can no longer upload solutions, only the expected output. The system uses the Java Error Correction Algorithm (JECA) that evolved from the algorithm used for b-type functionality. Besides trying to autocorrect the student's solution it generates a number of permuted parse trees in order to find out what the student was trying to do.

The further development of JITS also involved constructing an engaging, complete and accessible learning environment. An authoring tool for lecturers was added to submit new exercises by specifying the problem statement and description, required output and a code template. Considerable attention was also given to hint generation. A hint-object in JECA consists of information on the location of the problem, a proposed solution, the confidence of this solution and the type of hint (keyword replacement, grammatical hint, logic error et cetera). The data in the hint-object is used to generate a hint message for the student.

JITS was tested in multiple qualitative studies with positive results from students and professors. JITS is easy to use for both students and instructors. However, hints are more directed at the syntax level and do not help the student in approaching a problem. Because the system only knows the required output, hints cannot be given on the strategy level.



Problem: (3 of 5) in Problem Set # 2 *(Topic: Java Statements)*
Write a program called Power Generator which calculates the result of a number multiplied by itself.

Program Specifications:
This program requires the use of a function. A skeleton structure of the solution is given. You need to declare the variable: result View the image for this problem.

Required Output:
Result = 10000

```
public class Power {
   public int powergen(int num) {
      return num * num;
   }
   public static void main(String [] args) {
      Power p = new Power();
```

```
double result
p.powergen(19);
```

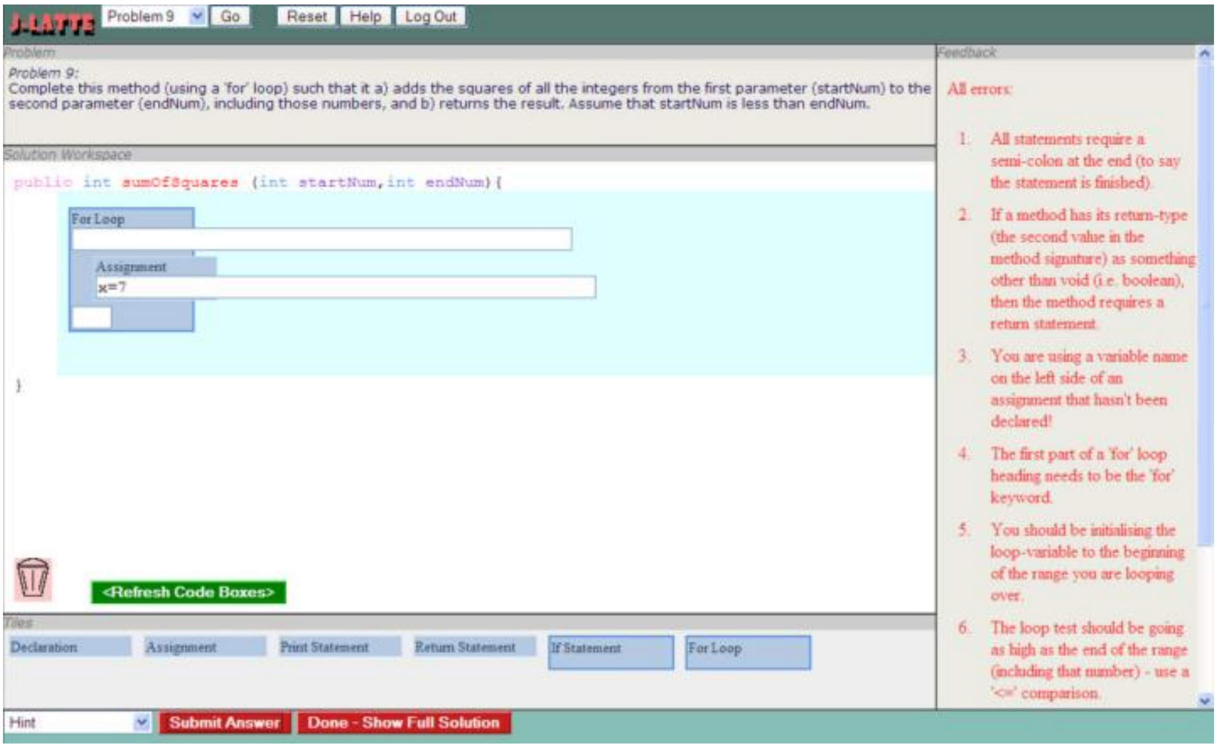
```
      System.out.println("Result = " + result);
   }
}
```

OUTPUT:

FIGURE 8 A FRAGMENT OF THE USER INTERFACE OF JITS

J-Latte. J-LATTE, the Java Language Acquisition Tile Tutoring Environment, is a Java tutor from 2009 (Holland, Mitrovic, & Martin, 2009). The tutor presents the student with simple programming exercises that can be solved with a subset of the Java programming language. The problem can be solved in two modes that can both be seen in Figure 9:

- *Concept mode.* Concepts are programming artefacts defined at a higher level, such as declaration, return statement and for loop. The student can select predefined concepts from the user interface and combine them to create the structure of the solution.
- *Coding mode.* When a student selects a concept, the accompanying code can be entered.



Problem 9:
Complete this method (using a 'for' loop) such that it a) adds the squares of all the integers from the first parameter (startNum) to the second parameter (endNum), including those numbers, and b) returns the result. Assume that startNum is less than endNum.

```
public int sumOfSquares (int startNum,int endNum){
    x=7
}
```

All errors:

1. All statements require a semi-colon at the end (to say the statement is finished).
2. If a method has its return-type (the second value in the method signature) as something other than void (i.e. boolean), then the method requires a return statement.
3. You are using a variable name on the left side of an assignment that hasn't been declared!
4. The first part of a 'for' loop heading needs to be the 'for' keyword.
5. You should be initialising the loop-variable to the beginning of the range you are looping over.
6. The loop test should be going as high as the end of the range (including that number) - use a '<=' comparison.

FIGURE 9 THE INTERFACE OF J-LATTE

J-LATTE uses constraints to represent domain knowledge. These constraints describe features of the solution that can either be syntactic, semantic or style-related. Semantic constraints compare a proposed solution to the formal specification of the problem, for example:

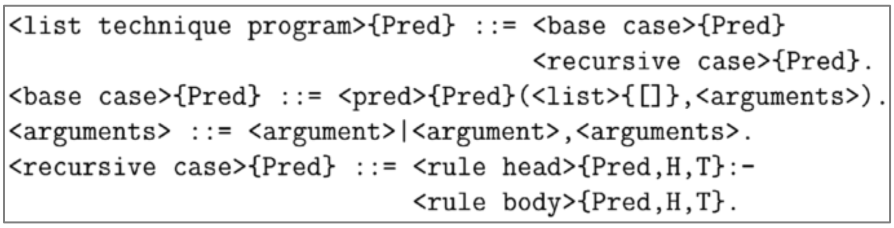
```
(sum-of-function-over-a-range :range (:from (method-arg :name "startNum") :to
(method-arg :name "endNum")) :function square)
```

This specification states that the *'sum-of-function-over-a-range'* pattern should be used, which could be any kind of loop, it specifies the lower and upper limit of the range and requires the presence of a square-function.

Feedback can be requested by the student at any time during the exercise. The system indicates whether the provided solution is correct or incorrect and also presents error messages related to the first or all constraints that are violated, with some hints on how to solve the errors. Using constraints does not force the student to follow a predetermined path in programming.

An experiment with students using the tutor showed promising results, although the number of participants was too low to draw any valid conclusions. J-LATTE supports the tackling of a problem in chunks by choosing building blocks and refining them step-by-step, but the feedback is not focused on following a particular solution strategy.

Prolog Tutor. The Prolog Tutor is a tutor that guides programming and performs error analysis for writing programs in the declarative, logic programming language Prolog (Hong, 2004). The underlying technique categorises Prolog programs that have the same programming technique or code pattern. These categorisations can be used for recognising and generating code. For each class of programs a set of grammar rules is determined (see Figure 10) and knowledge about the technique and coding is stored in *frames*.



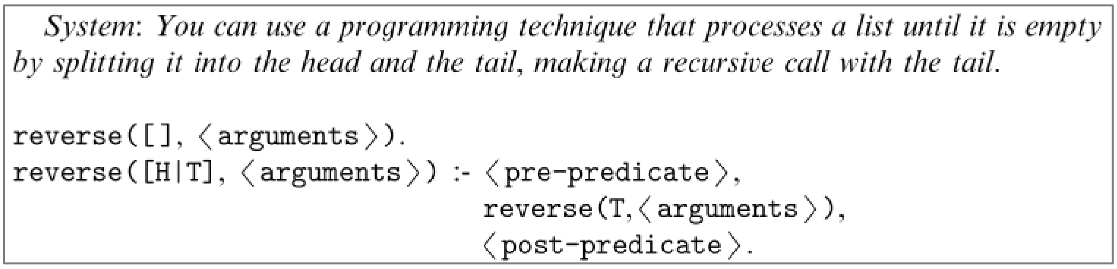
```
<list technique program>{Pred} ::= <base case>{Pred}
                                   <recursive case>{Pred}.
<base case>{Pred} ::= <pred>{Pred}(<list>{[]},<arguments>).
<arguments> ::= <argument>|<argument>,<arguments>.
<recursive case>{Pred} ::= <rule head>{Pred,H,T}:-
                           <rule body>{Pred,H,T}.
```

FIGURE 10 SOME PROLOG PROGRAMMING TECHNIQUE GRAMMAR RULES

Prolog programs can be parsed using these grammar rules. There can even be a hierarchy of programming techniques so the tutor can operate on different levels of abstraction. The instructor has to provide all possible solutions to an exercise. Differences in the order of clauses or predicates can be ignored by the system by applying certain rules.

A student working with exercises in the tutor can get help in two ways:

- Guided programming. The right programming technique is offered by providing a template the student can fill in. First, the template at the highest level of abstraction is offered. If the student asks for more help, a more detailed or specialised template can be given, as shown in Figure 11. The students might be asked to choose a specific technique if there are multiple approaches.
- Error analysis. If the system recognises that a student is using the correct programming technique, the tutor can detect if this technique is used in the right or wrong way and provides a detailed error message. Errors are treated one by one until the student solves them all. This is done by comparing the parsed student program to the parsed solution.



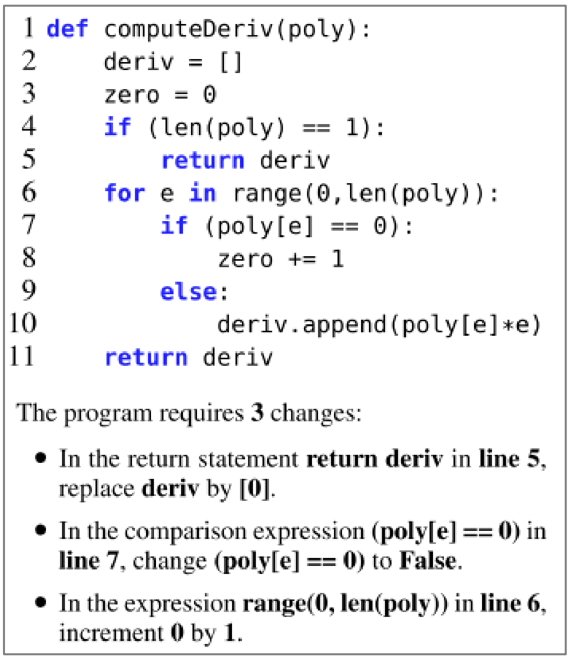
System: You can use a programming technique that processes a list until it is empty by splitting it into the head and the tail, making a recursive call with the tail.

```
reverse([], ⟨arguments⟩).
reverse([H|T], ⟨arguments⟩) :- ⟨pre-predicate⟩,
                               reverse(T,⟨arguments⟩),
                               ⟨post-predicate⟩.
```

FIGURE 11 THE RESPONSE OF THE PROLOG TUTOR TO A STUDENT ASKING FOR HELP

The tutor was tested with over a hundred solutions to one programming problem. Overall the tutor performed well, 98% of the correct programs and 95% of the incorrect programs were correctly diagnosed. The recognition of multiple solution strategies and the top-down approach of solving a problem are similar to the strategies for solving a problem in the Ask-Elle tutor.

Python tutor. The work of Singh et al. focuses on generating feedback for solutions to introductory programming problems in a large subset of Python (Singh, Gulwani, & Solar-Lezama, 2013). Feedback is provided after a complete solution is submitted and consists of the location of the error, the problematic expression and the modification, as shown in Figure 12.



```
 1 def computeDeriv(poly):
 2     deriv = []
 3     zero = 0
 4     if (len(poly) == 1):
 5         return deriv
 6     for e in range(0,len(poly)):
 7         if (poly[e] == 0):
 8             zero += 1
 9         else:
10             deriv.append(poly[e]*e)
11     return deriv
```

The program requires **3** changes:

- In the return statement **return deriv** in **line 5**, replace **deriv** by **[0]**.
- In the comparison expression **(poly[e] == 0)** in **line 7**, change **(poly[e] == 0)** to **False**.
- In the expression **range(0, len(poly))** in **line 6**, increment **0** by **1**.

FIGURE 12 GENERATED FEEDBACK FOR AN ERRONEOUS SOLUTION

As input to the system, the instructor has to write a reference implementation and an error model using an error model language the authors designed. The error model consists of a set of correction rules that solve the mistake a student might make together with appropriate feedback messages in natural language. All possible programs based on these rules applied to the solution of the student are then searched to find the one that most closely matches the reference solution. This is done by translating the program with the correction rules into a *Sketch program*. Sketch is a software synthesis tool that can complete a partial code implementation so it behaves like a given specification. The Sketch synthesiser finds the solution and feedback is generated based on the applied correction rules.

The tool was tested on thousands of student solutions and was able to provide useful feedback on at least 64% of the incorrect solutions. Ten seconds on average were needed to calculate the feedback. A limitation of the tool is its inability to deal with student programs that have large conceptual errors. The tool also cannot deal with structural requirements such as enforcing students to use recursion in their solution.

Data-driven tutors. A different approach is the generation of feedback based on student data from the past. Jin et al. (Jin, Barnes, & Stamper, 2012) use linkage graphs to represent correct student solutions. A linkage graph is an acyclic graph consisting of nodes representing states and directed edges representing the order of the statements. The student can ask for a hint while doing an exercise. A linkage graph is then created and the closest match with an existing graph will be used to provide feedback on the next step or the correction of an erroneous step. Multiple existing student solutions should be available with the risk that a specific alternative to solve the exercise might not be recognised. There is also work to be done on the format of the feedback messages.

Rivers and Koedinger propose a method for generating feedback by a data-driven approach focusing on class 3 problems, applying it to Python programming (Rivers & Koedinger, 2013). It is based on the conception that a large collection of student programs should give valuable information on possible solutions and common errors. Graphs are used to represent all possible paths to a solution, which the authors call *solution spaces*, in which the edges are steps and the nodes solutions states. Equivalent solution states are recognised by transforming programs into a canonical form. The canonical form is achieved by first creating an abstract syntax tree from a program, renaming all variables and attempting all known normalizing transformations. The best next step is calculated from this graph, by considering possible correct solutions strategies that are close to the current solution state the student is in and by considering frequently visited states by other students. Feedback is then calculated based on the differences between the current and the next state.

An advantage of a data-driven approach is that the instructor does not have to provide much input. A disadvantage is that there might be unusual solution strategies in the data set that a teacher would normally not encourage.

Other tutors. Another tutoring system was created at the University of Split (Dadic, Stankov, & Rosic, 2008). A component of this system teaches program design skills. The instructor provides a model solution that will be stored in a tree structure consisting of goals and plans. If the student program does not match the model solution, the system will give an error message and will provide help on how to proceed on multiple levels. The analysis of students programs is based on artificial intelligence techniques, although the exact details of the implementation are unclear.

The programming tutor developed by Weragama et al. focuses on the teaching of web programming in PHP (Weragama & Reye, 2013). The tutor is still being improved. The solution to an exercise is defined using a set of predicates describing a goal, a number of constraints and conditions. If all predicates can be recognised in the student program, it is considered correct, if not, feedback is provided. The tutor is not focused on providing feedback during programming.

In a tutor for Prolog programming (N. Le & Menzel, 2008) constraints are used to describe the solution space of programming problems, considering ill-defined programming exercises. Constraints can either refer to a specific problem or can be general Prolog constraints. Feedback is generated based on constraints that are not met by the student solution.

Jurado et al. (Jurado, Redondo, & Ortega, 2012) have developed a system that gives feedback based on software metrics and test cases. The structure of the solution is checked by the fuzzy evaluation of a number of metrics, such as lines of code and cyclomatic complexity. The correctness of the solution is checked by running several test cases. The provided feedback is then based on the test case or metric that deviates from the results of the model solution.

M-PLAT, Multi-Programming Language Adaptive Tutor (Nunez, Fernandez, & Carretero, 2010), focuses on providing exercises that are suitable for the student, solution checking, documentation and collecting statistics on student performance in one environment. Solution checking is done by the expert module that uses black box testing and white box testing. The system provides output in the form of error messages.

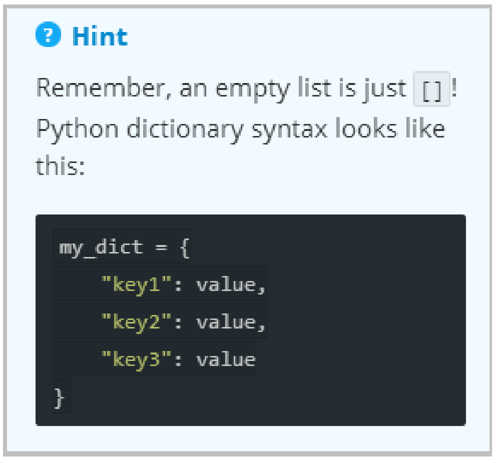
Commercial and online tutors. Beyond the scientific world quite a number of online programming tutors have emerged recently, such as Code School[7], Codeacademy[8] and Khan Academy[9]. Commercial parties also provide tutors, such as My Programming Lab[10] from educational publishing company Pearson. We do not know the technology behind these solutions and are only able to look at what they offer. Many free online tutors focus on absolute beginners and have a restricted set of exercises. Hints are often more general (Figure 13) or a template of the wanted solution that should be

---

[7] https://www.codeschool.com
[8] http://www.codecademy.com
[9] https://www.khanacademy.org
[10] http://www.pearsonmylabandmastering.com/northamerica/myprogramminglab

All retrieved April 25, 2014

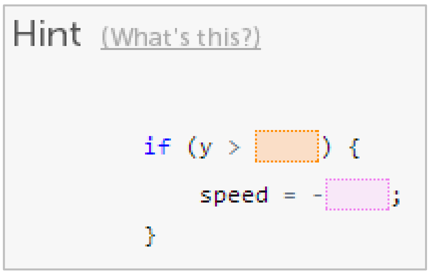
filled in (Figure 14). Hints from My Programming Lab (Figure 15) can be very cryptic when only submitting a small part of a solution.



Remember, an empty list is just `[]`! Python dictionary syntax looks like this:

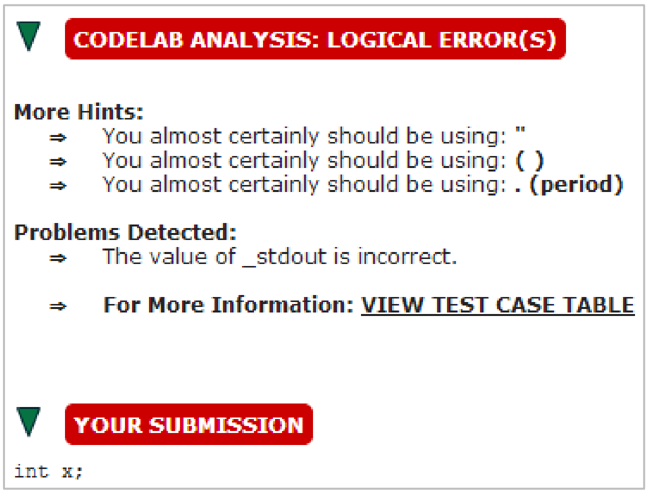
```
my_dict = {
    "key1": value,
    "key2": value,
    "key3": value
}
```

FIGURE 13 CODEACADEMY HINT



```
if (y >       ) {
    speed = -       ;
}
```

FIGURE 14 KHAN ACADEMY HINT



More Hints:
- ⇒ You almost certainly should be using: **"**
- ⇒ You almost certainly should be using: **( )**
- ⇒ You almost certainly should be using: **. (period)**

Problems Detected:
- ⇒ The value of _stdout is incorrect.
- ⇒ **For More Information: VIEW TEST CASE TABLE**

```
int x;
```

FIGURE 15 MY PROGRAMMING LAB FEEDBACK

3.4 PROGRAM ASSESSMENT TOOLS

An essential and both challenging aspect of programming tutors is the automatic assessment of student programs. A tutor should be able to determine if a candidate solution is either correct or incorrect accompanied by a list of semantic errors in the latter case. Semantic variations of a solution should be recognised and marked as correct.

In a fairly recent review of automatic assessment tools for programming developed from 2006 to 2010 (Ihantola, Ahoniemi, Karavirta, & Seppälä, 2010) two categories are identified. The first category is the use of industrial testing tools and the second 'various specialised solutions', subdivided into the traditional and widely used comparing of output, scripting and experimental approaches. The authors state that a very large number of tools are available, often with similar features and they plead for combining effort and disclosing implementation details.

The assessment tools that are based on test execution only are unable to consider the actual solution strategy making it impossible to locate and diagnose errors. Another approach is to assess programs using a number of metrics, such as lines of code and counting certain expressions or constructs, which has similar limitations. This paragraph focuses on a number of relevant automatic assessment approaches that have their focus on program semantics instead of testing and metrics.

More advanced assessment tools can be divided into three categories. The assessment method used by Gerdes et al. (Gerdes, Jeuring, et al., 2010) is an example of the *source-to-source* approach. Programs are assessed based on their equivalence with a solution from a set of generated solutions based on the strategy derived from model solutions. Syntactic differences are eliminated by program transformations. Other variants are *source-to-specification* and *specification-to-specification* in which a specification is a high-level description of goals in a program. These variations are not widely applied due to a number of difficulties.

Another example of source-to-source is the work on the assessment tool SIPLeS-II for Smalltalk programs (Xu & Chee, 2003). The tool uses a *transformation-based approach* to detect semantic errors in student programs and recognise many semantic variations. The student program is matched with a model program after they are normalised using various program transformations. Thirteen different semantics-preserving variations are identified, such as syntax variation, control structure variations and statement orders. All of these variations are taken into account and represented by transformation rules. The student program is first represented as an Abstract Syntax Tree and later as an Augmented Object-oriented Program Dependence Graph (AOPDG). An AOPDG is a flow graph that represents the operational semantics of a program. The tool can also be used for other programming languages and paradigms and has proven to be successful in an experiment with assessing student programs.

A similar approach is applied by Wang et al. (Wang, Su, Wang, & Ma, 2007), who developed an automatic grading system for programming exercises in C, called *semantic similarity-based grading*. The input to the grading system is a set of model solutions that represent various algorithms. A student program is correct if it is equivalent to one of these model solutions, meaning that they should have the same representation in the form of a system dependence graph. Before comparing them, the student graph and model graph are standardised using a number of semantics-preserving transformations to eliminate syntactic differences. If a match is found, values for matching size, structure and statements are calculated and combined into a semantic similarity value between 0 and 1. This number is used to indicate the differences, a full score of 1 means equivalence and therefore a correct solution, and can be converted into a grade.

The same conception of normalizing a program by various transformations and creating graphs representing the structure of the code is used by Naudé et al. (Naudé, Greyling, & Vogts, 2010) and Li et al. (Li et al., 2010). The assessment method from Naudé et al. is different because it compares the student graph to previously marked submissions of different quality from either other students or an instructor.

Multiple methods can be combined to improve assessment quality. Vujošević-Janičić et al. (Vujošević-Janičić, Nikolić, Tošić, & Kuncak, 2013) use automated bug detection, testing and control flow graph similarity to a model solution to assess programming exercises. A low-level intermediate representation of the program is used so the assessment methods can be applied to programs written in various languages.

3.5 CONCLUSION

Several observations can be deduced from the preceding research:

- The research area of intelligent tutors for programming was quite popular in its early days. Partly because of its complexity, this popularity faded. Recently there seems to be a modest revival by the emergence of a number of new (web-based) tutors.
- Many tutors can only provide feedback on complete student solutions and thus are not able to guide the student on their way to the right solution. Tutors that are able to provide feedback on incomplete solutions are often restricted to one solution strategy (class 2 problems).
- Not many tutors are widely available. Tutors that are still in development are usually only available within a university.
- A large number of program assessment tools mostly use testing techniques. Recent tools increasingly take a different approach by considering the semantics of a program.

How do the Ideas framework and the Ask-Elle tutor fit in this landscape of different tools and approaches? The combination of dealing with incomplete solutions to ill-defined (class 3) programming problems is rarely shown. The use of model solutions, the normalisation of programs and the matching of solution and student program are also used in other programming tutors and assessment tools. The ability to annotate model solutions with additional instructions is not seen elsewhere.

4 RESEARCH DESIGN

This research has been designed based on the problem description from Chapter 1 and the results of the literature study on existing programming tutors and the Ideas framework. We have concluded that adaptable tutors that deal with incomplete solutions for ill-defined problems can hardly be found for the imperative programming paradigm. In this chapter we state the research questions, the requirements and scope and list the validation questions.

4.1 RESEARCH QUESTIONS

In this thesis we elaborate on the answer to the following question:

How can we generate adaptable feedback to guide a student step by step towards a solution for an introductory imperative programming problem that can be solved by multiple strategies?

To find an answer to this question, a prototype for a domain reasoner for imperative programming has been implemented using the Ideas framework, consisting of the following components:

- An abstract syntax, a parser and a pretty printer for imperative programs.
- A set of rules.
- A generator for generating strategies by combining rules.
- A recogniser for equivalent and similar programs.
- Services for diagnosing programs and feedback generation.

A web based front-end for doing exercises has been created to provide an interface to the services of the domain reasoner.

By designing and developing the prototype we are able to formulate an answer to the following set of sub-questions:

i. What are the differences and similarities between the domain of imperative programming and the domains that have already been implemented using the Ideas framework?
ii. How do we construct a strategy for solving an imperative programming exercise?
iii. How do we represent incomplete imperative code?
iv. How do we distinguish different solutions to an imperative programming problem and when are solutions similar?
v. How can we recognise a strategy in (incomplete) imperative code?
vi. How can we generate semantic feedback for (incomplete) imperative programs?
vii. How can feedback be adapted by an instructor?

4.2 REQUIREMENTS AND SCOPE

Functional requirements. The tutor has the following functional requirements, described as user stories:

- As an instructor, I want to add a set of model solutions to a programming problem.
- As an instructor, I want to provide further instructions so I can adapt the feedback for a particular exercise.
- As a student, I want to ask for feedback so that I will know if I am on the right track.
- As a student, I want to ask for a hint so that I will know how to proceed.

The feedback and hints that are generated are at the strategy level. Besides semantic errors, students make many syntactical mistakes. These mistakes can often be identified by a compiler and are not included in the prototype. Generation of feedback can be provided on several levels. For absolute beginners it may be necessary to guide the student step-by-step through the construction of a while loop. For more advanced students, it may be enough to only indicate that a looping structure must be introduced. Since we do not build up a student model, we do not know at which level a student is when he or she is working on a programming problem. This issue can be left to the instructor who is able to annotate a model solution with directions on the level of feedback that should be generated. Another future option is that students indicate their level themselves. If the tutor is incorporated in a full learning environment in which student models are created, these models can be used to determine the skill level of the student.

Programming language. The prototype supports a well-known existing programming language. With this choice a large audience can be reached and in the long term we can combine our tool with existing programming tools to enrich the learning experience. In 2011 the results of a survey on practices in introduction courses on programming for undergraduates in the United States were published (Davies, Polack-Wahl, & Anewalt, 2011). Java as the main programming language appeared to be by far the most popular choice. The same result emerged from a similar study conducted in Australia in late 2010 (Mason, Cooper, & Raadt, 2012). Because of the popularity of Java both as a language taught in schools and universities and in the rankings shown in Chapter 1, Java is the language of choice. In addition to this, Java is taught in the first year of the IT-studies at Windesheim University of Applied Sciences where we are able to collect student data for validation. Support for PHP programs is also added because this language is used as a first programming language at Windesheim and is an additional source for data collection. Adding support for PHP also provides us with an opportunity to investigate the issues with developing a tutor that supports multiple imperative languages.

An imperative programming language such as Java has many language constructs and features. The full language is not supported in the prototype. We have selected a subset of the language including the constructs that are well-known and frequently used by novice programmers.

Exercise type. We focus on exercises for novice students who are learning to program. Exercises should be small; we only consider code fragments and methods that solve a single problem. We do not require fully executable code containing a main-function or a class definition. However, the exercises could be solved using multiple strategies or algorithms, corresponding to class 3 problems in the classification by Le et al. of educational problems based on their degree of ill-definedness (N.-T. Le et al., 2013).

Limitations. Besides implementing a reduced set of programming constructs, we also allow limited variety in expressions. We do want to be able to recognise variants of an expression, but the number is restricted. Compile errors are not taken into account. We assume that all programs that will be analysed do not contain any errors a compiler could identify.

4.3 VALIDATION

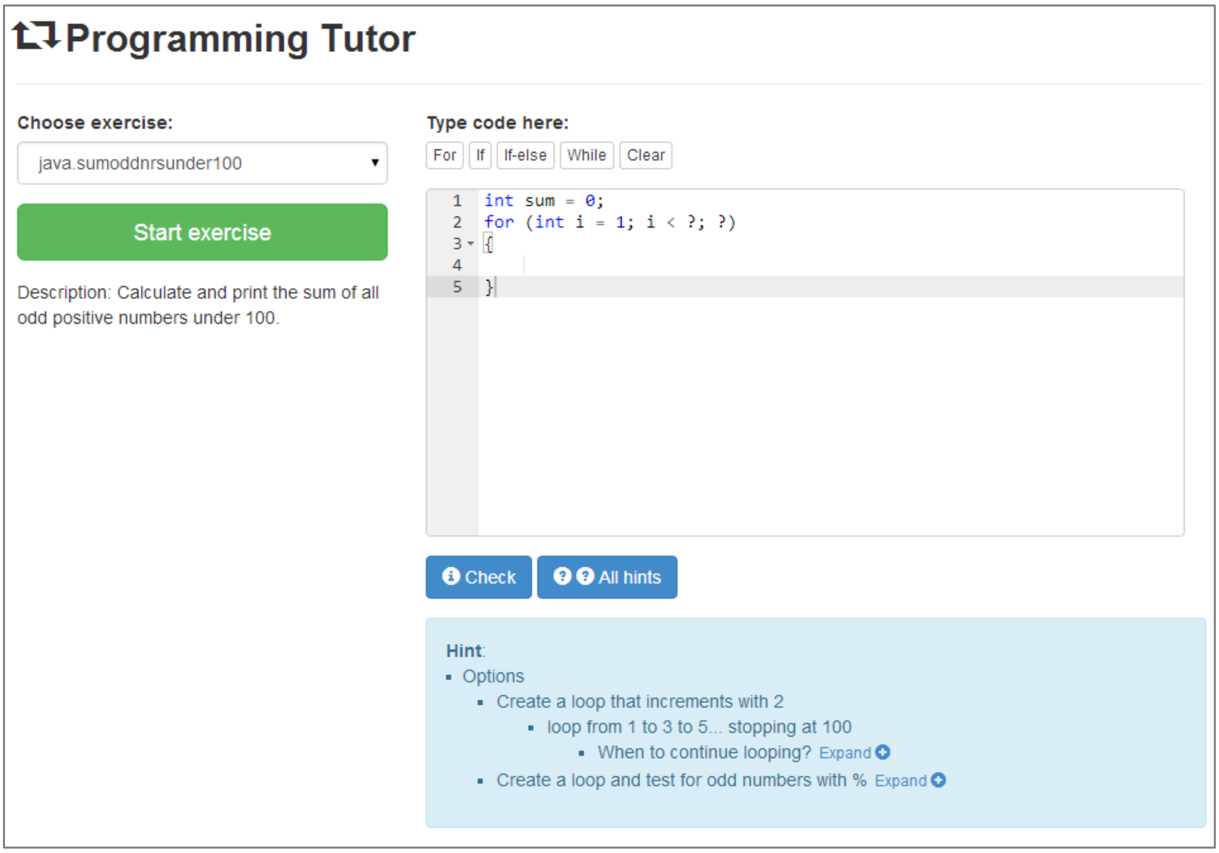
The main questions for the validation of the results are:

a. Are student programs that do or do not follow a known strategy recognised as such?
b. Can a step (or multiple steps) in a student program be recognised as either following a known strategy or not?
c. Do the generated hints lead the student to a solution?
d. Does the generated feedback reflect the annotations in model solutions?

The answers to these questions have been found by setting up various test cases, tutoring scenarios and analysing student data. Data was collected from actual students doing programming exercises.

5 A DOMAIN REASONER FOR IMPERATIVE PROGRAMMING

We have created a prototype of an imperative programming tutor, consisting of a domain reasoner for imperative programming and a user interface. The domain reasoner retrieves programming exercises from local files. We can work with the tutor using a basic command line interface. In addition to this, a web interface has been created that offers a simple learning environment in which students can solve exercises using the various feedback services. The feedback services are offered as web services with which we can interact through JSON or XML messages. Figure 16 shows a screenshot of this web front-end that has been created using HTML, JavaScript/JQuery and Ajax-calls with JSON messages to the services of the domain reasoner.



Choose exercise: java.sumoddnrsunder100

Description: Calculate and print the sum of all odd positive numbers under 100.

```
int sum = 0;
for (int i = 1; i < ?; ?)
{

}
```

Hint:
- Options
  - Create a loop that increments with 2
    - loop from 1 to 3 to 5... stopping at 100
      - When to continue looping? Expand
  - Create a loop and test for odd numbers with % Expand

FIGURE 16 WEB FRONT-END

The domain reasoner has been built using the Ideas framework, providing support for defining exercises, creating strategies and calculating feedback. In this chapter we elaborate on the components necessary to implement the domain reasoner for the domain of imperative programming and provide an answer to the various research question stated in the previous section. Knowledge of functional programming is required to fully understand the code fragments. This chapter is structured as follows:

- In Section 5.1 we discuss the internal representation of imperative programs. We have designed an abstract syntax and implemented a parser that translates program code in an imperative language into this abstract syntax. We also provide a pretty printer to show parsed programs in their original syntax. Because we allow programs to be incomplete, we describe how these programs can be represented, answering research question iii.

- In Section 5.2 the generation of a programming strategy from model solutions is explained, answering research question ii. To create a strategy we have defined a set of rules to represent steps in solving an imperative programming exercise. In our strategy generator we combine these rules into a strategy representing the various possible solutions and the paths to arrive at a solution.
- The issues concerning the recognition of student solutions can be found in Section 5.3. Variation in imperative programming code is a major issue. We need to recognise equivalent and similar programs using evaluation and normalisation of programs. This section focuses on research question iv.
- Finally the generation of feedback is examined in Section 5.4. We examine which services from the framework we can use to diagnose student programs and provide hints, and where we have to make adjustments. We also look into various ways an instructor can adapt feedback using specific settings or annotations. Research questions v to vii are addressed.

Research question i is addressed throughout the entire chapter.

5.1 REPRESENTATION OF IMPERATIVE PROGRAMS

The prototype can be used for languages that support the imperative programming paradigm. The various language constructs that are currently supported are shown in Table 8. The prototype does not support more advanced constructs such as methods declarations, object orientation, exception handling and object types. Less frequently used statements (switch, do while), data types such as chars and floats and various other operators have not been included in the prototype either.

LANGUAGE CONSTRUCT
Variable declarations (integer, boolean, string, array)
Variable initialisation
Integer, boolean, string and array literals, null value
Arithmetic operators: +, - (binary and unary), /, *, %
Comparison operators: >, <, <=, >=, !=, ==
Assignment operator: =, +=, -=, /=, *=, %=
Logic operators: &&, ||, !
Postfix/prefix operators: ++, --
String concatenation
Print statement
If statement, if-else statement
Loop statements: for, while
Calling library methods
Array access
Branching: break, continue

TABLE 8 SUPPORTED LANGUAGE CONSTRUCTS

Abstract syntax. An abstract syntax has been designed to represent imperative programs, supporting the language constructs of Table 8. This abstract syntax is implemented using data types in

Haskell, in which a `Program` is the top-level type that consists of zero or more statements. The `Statement` data type represents the various program statements:

```
data Statement =
        Block           [Statement]
    |   If              Expression Statement
    |   IfElse          Expression Statement Statement
    |   While           Expression Statement
    |   For             ForInit [Expression] [Expression] Statement
    |   Print           Expression
    |   VarDeclarations  DataType [Expression]
    |   ExprStat        Expression
    |   Empty
    |   Break
    |   Continue
```

Many statements are composed of one or more expressions, represented by the `Expression` data type:

```
data Expression =
        Infixed         InfixOp Expression Expression
    |   Assignment      AssignOp Expression Expression
    |   Prefixed        UnaryOp Expression
    |   Postfixed       UnaryOp Expression
    |   LiteralExpr     Literal
    |   IdExpr          Identifier
    |   Call            Identifier [Expression]
    |   Property        Identifier Identifier
    |   NewArray        DataType Expression
    |   ArrayAcc        Identifier Expression
```

The details of the other data types that are used, such as `Identifier`, `DataType` and `Literal`, are omitted. The abstract syntax is intentionally not very strict and specific, so various different languages can be represented by this data structure. An advantage of this general internal structure is that we can create programming exercises with solutions in one specific language that can also be solved using a different programming language.

Parser. Program code from a specific imperative language should be transformed into this internal representation using a parser. Language constructs that are very specific for a certain language are converted into a more general structure in the parsing process. For instance, some languages only allow one condition in a for statement, whereas other languages allow multiple conditions separated by a comma. The presence of a compiler or interpreter is always assumed so the parser does not have to perform semantic checks, such as type checking and object binding. If a program cannot be parsed, the tutor is not able to deal with it and the student should repair the code first based on compiler messages.

Currently there are parsers for two different programming languages: Java and PHP. Java is a well-known and widely used object oriented programming language that is often taught in schools and universities. PHP is a server-side scripting language that is usually embedded into HTML. The parsers have been implemented using the Parsec library[11] that provides a large number of parser combinators

---

[11] http://hackage.haskell.org/package/parsec, retrieved November 13, 2013

to simplify the parsing process. The parsers are written in the applicative style. A lexer is automatically created using a language definition for Java or PHP defining their keywords and special characters. Using this lexer we can easily define parsers for the language constructs, such as the while statement in the following example. The functions `exprP` and `statP` are the parsers for an expression and a statement respectively:

```
whileP :: Parser Statement
whileP = While <$ reserved "while" <*> parens exprP <*> statP
```

Pretty printer. In the output of the tutor we want to show programs in their original syntax. To enable this, a pretty printer has been implemented that converts abstract syntax into a textual representation that corresponds to the syntax of the programming language that is used. The printer is implemented using the PPrint library[12] that is based on the pretty printing combinators described by Philip Wadler (Wadler, 1998). The pretty printer is implemented for both the Java and PHP language. The next example shows the while statement converted into the `Doc` data type that represents a pretty document that can be shown as text.

```
instance Pretty Statement where
    pretty (While e s) = text "while" <+> parens (pretty e) <$> nested s
```

Support for incomplete programs. Students who have not finished their program yet should be able to receive feedback on their partial solution. Statements can be omitted in imperative programming, which should not create problems with parsing the program. To further support students in creating a program step by step the question mark (‘?’) character can be used inside a statement to represent an expression that is yet to be completed. A few examples are:

```
int x = ?;
sum = ? + ?;
for (?; ?; ?);
while (x < ?);
```

The expression data type has been extended with a ‘hole’ constructor. An integer is used to uniquely identify a specific hole.

```
data Expression = … | HoleExpr LocationID
```

The addition of a new symbol in the programing language will of course cause problems because the compiler is unfamiliar with this symbol. This implies that students cannot rely on compiler messages when using holes in their programs. Instead they are referred to the tutor that is able to recognise the holes and help the student complete the statement before continuing with the remaining program.

Testing. The parsers have been tested by parsing a large number of source files that include the supported language constructs in various forms. We also use QuickCheck (Claessen & Hughes, 2000), a library for testing that automatically generates test cases attempting to falsify properties. The combination of a parser and pretty printer together should satisfy the following (simplified) QuickCheck property, stating that the pretty printed representation of a program should be parsed into a program that is equal to the original:

---

[12] http://hackage.haskell.org/package/wl-pprint-1.1, retrieved November 13, 2013

```
prop_parsePrettyPrintedProgram :: Program -> Bool
prop_parsePrettyPrintedProgram program = program == (parse . pretty) program
```

To generate random programs we provide instances of the `Arbitrary` class for statements, expressions and the other data types that are used. We also provide a separate generator for programs that do not contain any holes, such as model solutions. An example fragment that generates a statement by choosing one from a list of statements is shown next. These statements consist in their turn of other arbitrary components. To prevent the generation of structures that are nested too deeply, we use a ‘sized’ generator that recurses towards statements that do not include other statements. If the size integer `n` reaches zero, these nested statements are not included in the choice.

```
instance Arbitrary Statement where
    arbitrary = sized $ sizedStatGen True

sizedStatGen :: Bool -> Int -> Gen Statement
sizedStatGen holes n = oneof $ notNested ++ if n > 0 then nested else []
    where
        notNested =
            [
                Print <$> arbEx,
                VarDeclarations IntType <$> sizedVector assignExprGen (1,3),
                return Break,
                …
            ]
        nested =
            [
                If <$> arbEx <*> smallerStat,
                IfElse <$> arbEx <*> sizedBlock <*> smallerStat,
                …
            ]
        smallerStat = oneof [sizedStat, sizedBlock]
        counter     = makeIdt "i"
        arbEx       = exprGen holes
        sizedStat   = sizedStatGen holes $ n `div` 10
        sizedVector = (>=>) choose . flip vectorOf
        sizedBlock  = makeBlock <$> sizedVector sizedStat (1, 5)
```

To generate programs that do not deviate too much from real world programs, we have to further control the randomness. In the next example we show how a for statement is generated. To avoid the accidental creation of infinite loops, a counter variable is used that is initialised at a number between zero and ten, increments with one each iteration and ends at 99.

```
For
 <$> ForInitExpr . (:[]) . Assignment Assign counter . makeInt <$> choose (0,10)
 <*> pure [Infixed Less counter $ makeInt 99]
 <*> pure [Postfixed Incr counter]
 <*> smallerBlock
```

We control the generation of expressions by providing a distribution and using custom-made generators such as `assignExprGen` (for variable assignments) and `arithExprGen` (for basic arithmetic expressions) that generate common expressions. The `frequency` function randomly chooses one of the generators based on the distribution.

```
exprGen :: Bool -> Gen Expression
exprGen holes = frequency $
    [   (40, makeInt <$> choose (0, 999)),
        (40, IdExpr <$> arbitrary),
        (30, arithExprGen),
        (20, assignExprGen),
        (5, Call <$> arbitrary <*> vectorOf 2 (exprGen holes)),
        (5, ArrayAcc <$> arbitrary <*> exprGen holes)
    ] ++ [ (10, HoleExpr <$> arbitrary) | holes ]
```

An example of a randomly generated program is shown below. Note that there are some language constructs that will not be accepted by a Java compiler. We do not check for correct declaration and initialisation of variables. There are many possible improvements; however, for testing purposes these programs are adequate.

```
continue;
int z = y != z;
while (true)
    continue;
for (i = 8; i < 99; i++)
    print (211);
for (i = 2; i < 99; i++)
{

    if (658)
    {
        continue;
    }
    if (161)
        y(599, 921 < 225);
    break;
}
print (x);
```

5.2 STRATEGIES FOR IMPERATIVE PROGRAMMING

To use the Ideas framework for calculating feedback, we need to specify a strategy for each exercise. In an educational setting, the instructor serves as a guide to show students how to program. When an instructor is not present, we would like to stay close to what an instructor would have said when a student asks for help. Therefore model solutions from an instructor are used as a basis to provide feedback. This approach is also used in a number of other recent programming tutors (Dadic et al., 2008; Gerdes, Jeuring, et al., 2012; Hong, 2004; Singh et al., 2013) and provides a number of advantages:

- An instructor can easily add new exercises.
- Models programs can be annotated, providing extra opportunities for didactic guidance. Annotating model solutions is elaborated in Section 5.4.3.

Potential difficulties that should be looked into are the large solution space, which is discussed in the following sections, and the lack of clarity on what exactly distinguishes one solution from the other. It is the instructor's responsibility to provide model solutions that represent the solution space of an exercise. Every model solution should preferably represent a different algorithm that solves the problem. But what separates one algorithm from the other? This issue is addressed in Section 5.3.1.

The strategy to work towards a particular model solution should reflect how imperative programs are implemented. Imperative programs can be constructed in several ways:

- Quickly constructing a coarse solution and then refactoring it until there are no errors left. This approach might reflect the trial and error style students often adopt.
- Programming by contract: defining pre- and post-conditions prior to the actual implementation (Dijkstra, 1975).
- The stepwise decomposition of a program using refinement steps (Wirth, 1971).
- Building up a program line by line, manipulating the program state in the meantime.

In recent tutors that support incomplete programs, we recognise the third option for an imperative language (Holland et al., 2009) and logic programming (Hong, 2004). In the Ask-Elle tutor for functional programming (Gerdes, Jeuring, et al., 2012) refinement steps are used to gradually make a program more complete by replacing unknown parts (holes) by actual code. The last option is used in two data-driven tutors for imperative programming (Jin et al., 2012; Rivers & Koedinger, 2013), although the steps are generally larger than just one line.

An advantage of the last option is that the compiler can provide help in most situations, for example variables are always declared before they are used. We have selected this style for the tutor, because of this advantage together with the ability to help the student from start to finish and not only after creating a first solution entirely on their own. We also incorporate refinement for composed language constructs.

To create a strategy using the Ideas framework we need two components that are elaborated in the next sections:

- Rules that represent the steps a student can take to gradually build up a solution.
- A strategy generator that generates a strategy from model solutions using these rules.

5.2.1 RULES

A strategy to solve an exercise is made up of a number of steps, or ‘rules’. The Ideas framework enables the creation of rules based on a transformation function. Two types of rules are used in the tutor for imperative programming: *append rules* and *refinement rules*, which will be explained in more detail in this section.

Append rules. An append rule appends a statement to the end of a block, which corresponds to updating the program state line by line. An example of three consecutive applications of an append rule is:

```
x = 5;
```

⇨

```
x = 5;
y = 7;
```

⇨

```
x = 5;
y = 7;
avg = (x+y)/2;
```

⇨

```
x = 5;
y = 7;
avg = (x+y)/2;
print(avg);
```

Suppose we have a program with multiple nested statements, such as:

```
if (x > 10) {
    for (i = 0; i < x; i++) {
        print(i);
    }
}
```

It is unclear where a new statement should be appended. There are three options: after the print statement inside the for statement, after the for statement inside the if statement or after the if statement. To identify the specific location of an append rule, we extend the construction of a `Block` with an integer that uniquely identifies this block. To enable adding statements to the highest level of the program, every program will be parsed into a program with a block at top level.

```
data Statement = … | Block LocationID [Statement]
```

An append rule can be created using the `appendStat` function as shown below. The `pref` integer is used for rule ordering, which is explained at the end of this section. The rule can only be applied if the block with the specified identifier is present in the program exactly once. We use the `transformBi` function from the Uniplate library (Mitchell & Runciman, 2007) for generic traversals. The library provides functions to easily traverse and manipulate complex data structures to avoid writing a lot of repetitive, ‘boilerplate’, code. The Biplate variant (`transformBi`) is used because the `Program` data type combines multiple other data types. The transformation `append'` is applied to all statements in a program, including nested statements that can be found inside statements such as while and if. If the unique identifier of a block equals the location parameter, the new statement is appended to the block.

```
appendStat :: Statement -> LocationID -> Int -> Rule Program
appendStat newStat loc pref = makeRule ruleId $ append
    where
        append p
            | nrOfBlocksById loc p == 1 = Just $ transformBi append' p
            | otherwise                 = Nothing

        append' (Block i stats) = Block i $ stats ++ [newStat | i == loc]
        append' stat = stat

        ruleId = …
```

Refinement rules. A refinement rule replaces a hole by an expression. An example of applying a sequence of refinement rules is:

`avg = ?;` ⇨ `avg = ? / ?;` ⇨ `avg = sum / ?;` ⇨ `avg = sum / 2;`

The code for creating a refinement rule is similar to the code for the append rule, apart from the implementation of the transformation function `refine`' that replaces the hole with the new expression:

```
refineExpr :: Expression -> LocationID -> Int -> Rule Program
refineExpr newExpr loc pref = describe name . makeRule ruleId $ refine
    where
        refine p
            | nrOfHolesById loc p == 1 = Just $ transformBi refine' p
            | otherwise                = Nothing

        refine' e@(HoleExpr i)
            | i == loc  = newExpr
            | otherwise = e
        refine' e = e

        ruleId = …
        name = …
```

Other rules. We have defined a rule for inserting a statement at any location in a program; however this rule has not yet been used. Defining rewrite rules is also an option for future research, as described in Section 7.4.

Rule ordering. Rule ordering is used to give preference to certain model solutions and language constructs. We define a rule ordering based on an integer that is the suffix of the rule identifier, for example rule 'if-else-at-1.6' will be ordered using the suffix '6'. We define preference during the strategy generation process, which is described in the next section.

5.2.2 GENERATING STRATEGIES

Using the rules described in the previous section, we can now specify strategies for the stepwise development of a program. We have created a strategy generator that accepts a set of model programs as input and produces a strategy as output. A number of normalisations are performed on the program before the strategy generation. During the generation process we maintain a state that stores a counter and the feedback level (the usage of this level is elaborated in Section 5.4.3). The counter is used to uniquely number the blocks and holes during the generation of the strategy.

```
type GenState a = State (LocationID, Int) a
```

We define a `GenStrategy` class with a function that takes any value (and some additional parameters) and returns the current state and a corresponding strategy for a program. We provide implementations for the main data types `Program`, `Statement` and `Expression`.

```
type StrategyGenerator a = LocationID -> Int -> a -> GenState (Strategy Program)

class GenStrategy a where
    genStrat :: StrategyGenerator a
```

We also define a class of types for which we can generate a strategy together with a specific location, which could either be a hole (for expressions) or a block (for statements).

```
class GenStrategy a => GenStrategyWithLoc a where
    genStratWithLoc   :: Int -> a -> GenState (a, Strategy Program)
    genStratsWithLocs :: Int -> [a] -> GenState ([a], [Strategy Program])
    genStratsWithLocs = mapAndUnzipM . genStratWithLoc
```

The overloaded function `genStratWithLoc` is defined for both statements and expressions and returns a tuple with either a new hole or a new block with a unique number together with the strategy of the input. The `getNextNr` function returns the next available number and updates the counter in the state. We only show the code for the expression variant.

```
genStratWithLoc pref expr =
    do
        loc   <- getNextNr
        expr' <- genStrat loc pref expr
        return (HoleExpr loc, expr')
```

As mentioned before, we generate a strategy for each statement and expression in a program. We illustrate this by showing the implementation of a number of language constructs, but are far from complete.

If statement strategy. The next fragment shows the code for generating a strategy from an if statement.

```
genStrat loc pref (If condition body) =
    do
        (hole, condition') <- genStratWithLoc pref condition
        (block, body')     <- genStratWithLoc pref body
        app                <- appRule (If hole block)
        return $ app <*> condition' <*> body'
```

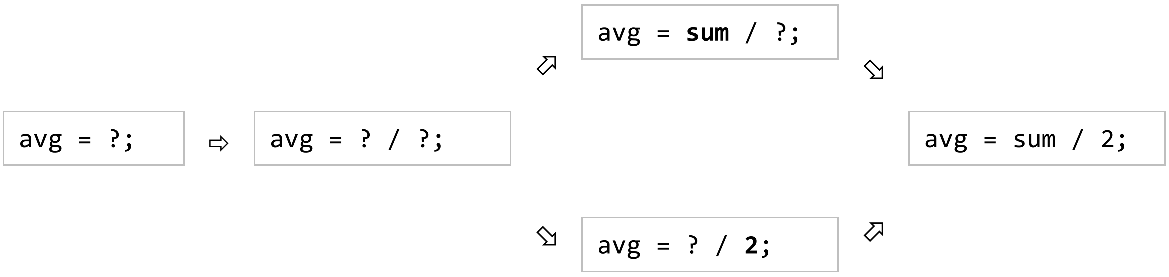
First, a hole is created for the condition in the if statement together with a corresponding strategy. Next, a block and the strategy for the body are generated, followed by an append rule for an empty if using the helper function `appRule`. The resulting strategy consists of a sequence of creating an empty if, building up the condition and finally creating the body using the sequence (`<*>`) combinator from the Ideas framework, as illustrated in the following example:

`if (?) {}` ⇨ `if (isOk) {}` ⇨ `if (isOk) { call(); }`

Infix expression strategy. The next fragment shows how a strategy is created for an expression, such as ‘(a + b) < 2’. The result is the introduction of an infix expression with holes on both sides of the operator, followed by the interleaving (`<%>`) of the sub strategies for the left and right operands of the expression. Refining the left hole first has a higher preference.

```
genStrat loc pref (Infixed op e1 e2) =
    do
        (hole1, e1') <- genStratWithLoc (pref + 1) e1
        (hole2, e2') <- genStratWithLoc pref e2
        return $ refRule (Infixed op hole1 hole2) <*> (e1' <%> e2')
```

This implies that we can arrive at an expression consisting of several sub expressions in multiple ways:



Loop strategy. A more complex situation arises when we encounter a for statement in a model solution. A for statement is easily transformed into a while statement, which we want to support in our tutor. In the corresponding code we create a strategy for the for statement together with an accompanying while statement and combine their strategies with the choice (`<|>`) combinator. The while statement is constructed by moving the initialisation of the for statement to a new statement preceding the while. The increment expression of the for statement is appended to the end of the body of the loop. However, we only include the strategy for a while if the for loop has exactly one condition (`length cond == 1`) to avoid an empty while condition. The details of the `forStrat` and `whileStrat` functions have been omitted.

```
genStrat loc pref (For forInit cond incr body) =
    liftM2 (<|>) (forStrat forInit cond incr body) optionalWhile

    where
        optionalWhile
            | length cond /= 1  = return failS
            | otherwise         = do
                -- convert to while
                let newCond = head cond
                    newInit = forInitToStat forInit
                    newBody = body <> mconcat (map ExprStat incr)
                init' <- genStrat newInit loc
                while <- whileStrat newCond newBody
                return $ init' <*> while
```

The resulting strategy allows both of the following sequences (skipping some intermediate states):

```
i = 0;
```

⇨

```
i = 0;
while(?)
{}
```

⇨

```
i = 0;
while(i < 8)
{}
```

⇨

```
i = 0;
while(i < 8)
{ i++; }
```

```
for(?;?;?)
{}
```

⇨

```
for(i=0;?;?)
{}
```

⇨

```
for(i=0;i<8;?)
{}
```

⇨

```
for(i=0;i<8;i++)
{}
```

Block strategy. We are faced with a challenge when we want to implement a strategy generator for a `Block`, a list of statements. Every program is a list of statements and inside composed statements such as loops we find nested lists of statements. In imperative programming a program can be developed line by line, continuously manipulating the program state. The order of some statements can be changed with no consequences for the output of the program, for example:

Model:

```
1  x = 5;
2  y = 7;
3  count = 2;
4  sum = x + y;
5  avg = sum/count;
6  print(avg);
```

Alternative:

```
y = 7;
x = 5;
sum = x + y;
count = 2;
avg = sum/count;
print(avg);
```

In this example lines 1 and 2 and lines 3 and 4 are switched with no consequences for the resulting program. The option to change the order of statements depends on a number of properties. For example, line 5 and 6 can never be switched because the value has to be calculated first before it can be printed. A relation has been defined to determine whether a statement depends on another preceding statement. Dependencies often arise when using or changing variables. We have created a class that locates identifiers that are changed or used in a particular data type and provide instances for statements and expressions.

```
class FindIds a where
    usesIds      :: a -> [Identifier]
    changesIds   :: a -> [Identifier]
```

To illustrate these functions we provide parts of their implementation. We start with the `usesIds` function on array accessors, such as list[i]. Both ‘list’ and ‘i’ are used in this expression.

```
usesIds (ArrayAcc idt expr) = idt : usesIds expr
```

Next we examine `changesIds` on a postfix expression such as counter++, where ‘counter’ will be identified as changed.

```
changesIds (Postfixed Incr (IdExpr i) = [i]
```

Using these instances we can now define the relation:

```
s2 `dependsOn` s1 =
        -- option 1
        (changesIds s1 ∩ (changesIds s2 ∪ usesIds s2)) ≠ ∅
        -- option 2
    ||  (usesIds s1 ∩ changesIds s2) ≠ ∅
        -- option 3
    ||  ([p | Print p <- universe s2] ≠ ∅ &&  [p | Print p <- universe s1] ≠ ∅)
        -- option 4
    ||  s1 == Break || s1 == Continue || s2 == Break || s2 == Continue
```

The first option identifies a dependency if a variable is changed in statement s1 that is used or changed in the statement s2, for example in:

```
x = 1;
print(x);
```

The second option checks if a variable is used in statement s1 that is changed in statement s2, for example in:

```
a = x;
x = 9;
```

Option three checks if both statements generate side-effects, in this case by containing a print statement. Because they make up the output of a program, their order cannot be changed. We show an example below. We currently do not include checking for side-effects in library functions.

```
print("first");
if (someBoolean)
    print("maybe second");
```

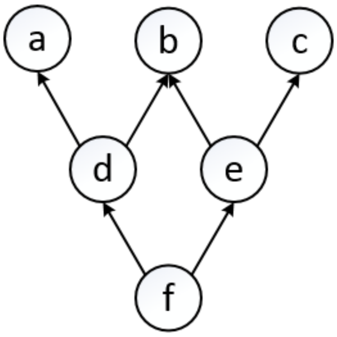
In the fourth option we identify a dependency if a control flow statement is encountered that we cannot move without consequences.

Using this relation, we are able to construct a *directed acyclic graph* (DAG) for a list of statements. A directed acyclic graph is a dependency graph without cycles. The arrows in the graph indicate that a statement depends on another prior statement, implying that no cycles can occur in the resulting graph. As an example, the following list of statements and the corresponding graph is shown:

Block:

```
a = 1;
b = 2;
c = 3;
d = a + b;
e = b + c;
f = d + e;
```

Dependency graph:



The following code converts a list of elements with dependencies into an intermediate list of nodes that can be used to construct a graph. First we pair each item in the list with a unique number and we reverse the list. We can now use a right fold to process the list from right to left with an empty list as a starting value. The `calcDeps` function will be applied to a node and a list of previous nodes and adds an adjacency list to this node with the preceding nodes that the node depends on.

```
list2nodes :: Deps a => [a] -> [NodeInfo a Int]
list2nodes list = foldr calcDeps [] numberedList
    where
        numberedList = reverse $ zip list [1..]

        calcDeps (item, nr) prevNodes = (item, nr, deps) : prevNodes
            where
                deps    = map snd3 allDeps
                allDeps = filter (dependsOn item . fst3) prevNodes
```

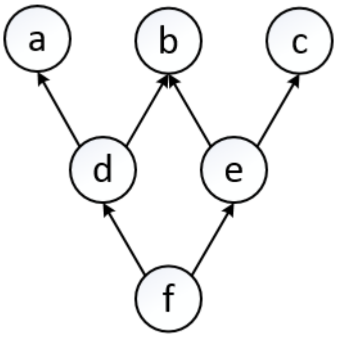
Now a strategy for each node can be generated and stored in the nodes, from which we can build a graph using the `Data.Graph` library.

```
list2stratGraph :: (Deps a, GenStrategy a) => LocationID -> Int -> [a] ->
    GenState (DependencyGraph (Strategy Program) Int)
list2stratGraph loc pref = liftM graphFromEdges . mapM nodesWithS . list2nodes
    where
        nodesWithS (stat, nr, dep) =
            liftM (\s -> (s, nr, dep)) $ genStrat loc pref stat
```

From this graph we can build a strategy by listing all topological sorts of the graph. A topological sort is a possible ordering of the vertices with the property that for every edge representing a dependency from a node ***a*** to a node ***b***, ***b*** precedes ***a*** in the ordering. The graph is converted into a left-factored strategy using the strategy combinators for sequence (`<*>`) and alternatives (the choice `<|>` combinator applied to multiple strategies). The corresponding code for building a strategy from a dependency graph is:

```
dependencyGraph :: IsStrategy f => DependencyGraph (f a) key -> Strategy a
dependencyGraph (graph, vertex2data, _) = g2s []
   where
      g2s seen
         | null reachables = succeed
         | otherwise       = alternatives $ map makePath reachables
         where
            reachables      = filter isReachable $ vertices graph \\ seen
            isReachable     = null . (\\ seen) . (graph!)
            makePath vertex = (fst3 . vertex2data) vertex <*> g2s (vertex:seen)
```

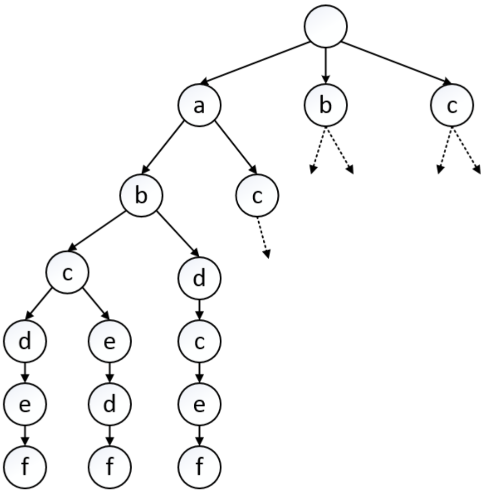
The original graph with the resulting strategy is shown next.



⇨

```
    (a <*> (    (b <*> (    (c <*> (
                                   (d <*> e <*> f)
                               <|> (e <*> d <*> f)))
                        <|> (d <*> c <*> e <*> f)))
            <|> (c <*> …)))
<|> (b <*> …)
<|> (c <*> …)
```

The strategy represents the following paths:



The conversion of a DAG into a corresponding strategy is a general solution that may be used in domains other than imperative programming. Therefore, the function has been added to the Ideas framework as a new strategy combinator.

Finally, we can use these functions to create the instance for `GenStrategy` for a list of statements.

```
instance (GenStrategy a, Deps a) => GenStrategy [a] where
   genStrat loc pref = liftM dependencyGraph . list2stratGraph loc pref
```

Exercise strategy. The final step in generating a strategy from a set of model solutions is combining the strategy from each model into one final strategy using the alternatives combinator. We summarise the complete strategy generation process in the following figure.

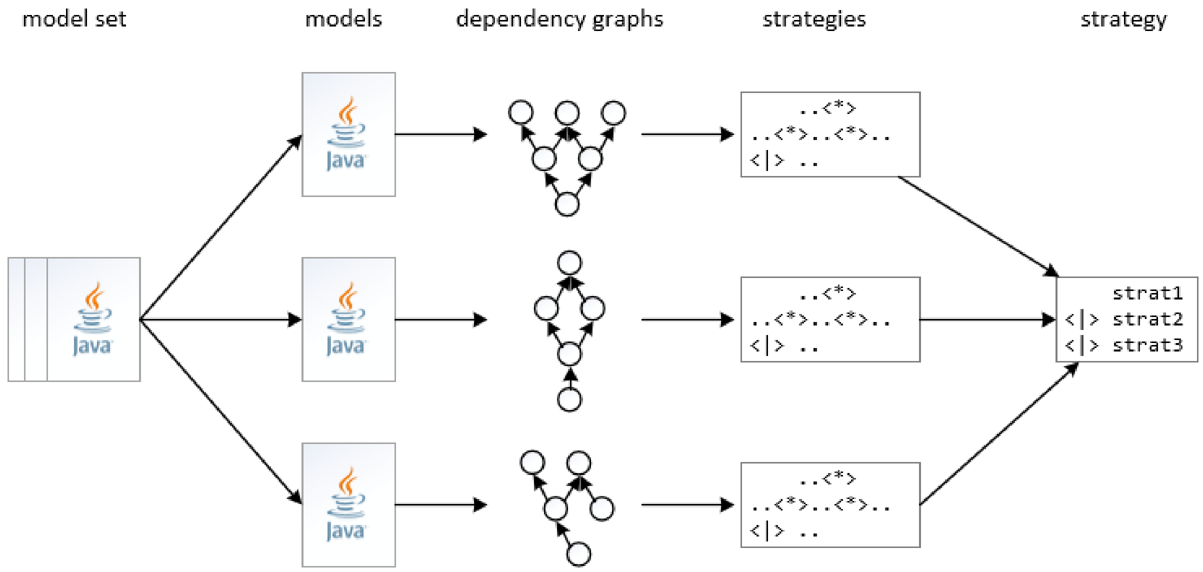


FIGURE 17 STRATEGY GENERATION PROCESS

5.3 RECOGNISING SOLUTIONS

From the strategy that is generated from model solutions, we can already recognise various solutions to an exercise: multiple algorithms, variation in statement orders and different language constructs. There are of course many more variations that we should incorporate. If a student works on a program creating a solution that closely matches a model solution but is not exactly the same, the student solution should be recognised. We need to further establish when a student program and a model solution can be considered 'equal'. We want to recognise if a student closely follows the solution and corresponding strategy that the instructor devised. Furthermore, if the student deviates from this strategy, we would still like to provide some response. The Ideas framework defines two relations on solutions that support the feedback generation: *similarity* and *equivalence*. In the next sections we elaborate on the implementation of these relations for the domain of imperative programming.

5.3.1 SIMILARITY

A model solution represents an algorithm to solve a particular programming problem. We have not yet established what exactly distinguishes one algorithm from another algorithm. Moreover, the definition of an algorithm lacks a clear answer and is subject to various interpretations. Blass et al. (Blass, Dershowitz, & Gurevich, 2009) argue against the notion that algorithms are equivalence classes of programs, implying that there is no 'precise equivalence relation capturing the intuitive notion of the same algorithm'. They provide several examples to illustrate their point, stating that opinions, subjective judgment and intended purpose influence this relation as well as a lack of clarity concerning the transitivity property.

We need to define our own relation on algorithms that indicates if two solutions are similar, to support the instructor in supplying a number of model solutions. The strategy to arrive at a particular model solution already incorporates a number of variations, for instance the ability to recognise that the order of two statements is irrelevant. These variations are recorded in the strategy, so the tutor is able to use them in the feedback process. Other than that, there are many more minor variations that we do not want to identify as different solutions. As an example, we consider the following two program fragments:

```
x = "*";
for(i = 0; i < 8; i++)
    print(x);
```

```
for(cnt = 1; cnt <= 8; cnt+=1)
{
    print("*");
}
```

In both solutions a looping structure is used to print a star symbol eight times. The differences (use of an extra variable, different loop counters, different variable names) do not change the algorithm used for this program and we would like to recognise the second program as a correct alternative for the first.

We need a similarity relation to further determine if two programs are similar when matching the student solution with a program derived from the exercise strategy. We define that two programs are similar if their representation in abstract syntax is equal, after *normalising* each program. Thus we can define the similarity relation ≈ as:

```
(≈) :: Program -> Program -> Bool
p1 ≈ p2 = normalise p1 == normalise p2
```

This relation is an equivalence relation, therefore the properties for symmetry, reflexivity and transitivity all hold. Similar programs have the same canonical form after normalization. Normalisation continuously attempts to perform a series of transformations until no more transformations can be applied. When transforming a student program, we should consider that the student might not have finished yet and that the program still contains holes. Certain transformations that can be performed on complete programs, such as dead code elimination, cannot simply be applied to incomplete programs. We should carefully consider which transformations can and cannot be applied to student programs.

Because our `Program` data type contains several elements that are irrelevant for the semantics of the actual program, such as annotations and location identifiers, we have defined the data type `BProgram` that is free from this overhead. For statements and expressions we have created new types. A class `ToBase` is defined to convert any data type into a corresponding `BProgram`.

```
class ToBase a where
    toB :: a -> BProgram
```

Instances are provided for `Program` and `BProgram`. Two `BPrograms` can easily be compared based on their structure. This conversion is performed prior to the various normalisations, implying that the normalisations are defined for the `BProgram` data type.

Xu and Chee (Xu & Chee, 2003) have identified 13 types of *semantics-preserving variations* (SPV). An SPV changes the computational behaviours (operational semantics) of a program while preserving the computational results (computational semantics). We incorporate some of these variations in different components of the tutor (the abstract syntax, the strategy) as shown in Table 9. Some differences that have not yet been captured in the abstract syntax or in the strategy are implemented in a normalisation procedure.

	DESCRIPTION	AST	STRATEGY	NORMALISATION
SPV1	Different algorithms		✓	
SPV2	Different source code formats	✓		
SPV3	Different syntax forms	✓	✓	✓
SPV4	Different variable declarations		✓	✓
SPV5	Different algebraic expression forms			✓
SPV6	Different control structures		✓	
SPV7	Different Boolean expression forms			✓
SPV8	Different temporary variables			
SPV9	Different redundant statements			
SPV10	Different statement orders		✓	
SPV11	Different variable names			✓
SPV12	Different program logical structures			
SPV13	Different statements		✓	

TABLE 9 SEMANTICS-PRESERVING VARIATIONS

We will now discuss the transformations from the categorisation by Xu and Chee. We show the variations for which we have provided a transformation function or a different solution in the parser or strategy. We have implemented a number of transformations, but are by no means complete. To increase the number of program variants that we can recognise, we would have to add several more transformations. However, some transformations have deliberately been omitted because the resulting program would deviate too much from the instructor solution.

SPV1 Different algorithms. We create a strategy for each model solution and combine them into one strategy.

SPV2 Different source code formats. Differences in source code format, such as whitespace and comments, are eliminated by the parser.

SPV3 Different syntax forms. Some variations in the syntactic form are eliminated by the parser, such as the multiple ways to declare an array in Java, implying that the following statements are similar:

`int list[];` ≈ `int[] list;`

We also parse the following similar statements into the same internal representation:

`int[] list = new int[] {1, 2, 3};` ≈ `int[] list = {1, 2, 3};`

Another syntactical difference is using braces when there is only one statement in the body of another statement, which is normalised in a transformation. As a result, the next two statements are similar:

```
if (x) f();
```

≈

```
if (x) { f(); }
```

Initialising an array together with the declaration or initialising the array element by element after declaration, as shown in the next example, are both allowed and are incorporated in the strategy. The first option, however, has a higher preference.

```
int[] x = {1, 2, 3};
```

```
int[] x = new int [3];
x[0] = 1;
x[1] = 2;
X[2] = 3;
```

SPV4 Different variable declarations. We perform two normalisations with regard to the declaration and initialisation of variables. We separate statements that contain multiple variable declarations, such as ‘int x, y’. Next, we separate declarations that include an initialisation of the variable, such as ‘int x = 0’. Combining these two normalisations, the following two programs are considered similar:

```
int x = 0, y = 1;
```

≈

```
int x;
x = 0;
int y;
y = 1;
```

Note that in the program on the right a different ordering of statements is allowed. This variation is taken into account by performing the two mentioned normalisations before generating a strategy for a model program.

SPV5 Different algebraic expression forms, SPV7 Different boolean expression forms. Several transformations are performed on the level of expressions using the `simplifyExpr` function. Operators that are used as syntactic sugar, such as ++, --, += et cetera are eliminated and replaced by an equivalent expression using basic operators. To compare integers, operators <=, > and >= are rewritten using only the < operator. We show a fragment of this function that transforms expression ‘x++’ into ‘x = x + 1’ and ‘x += y’ into ‘x = x + y’.

```
simplifyExpr :: Expression -> Expression
simplifyExpr (Postfixed Incr a)         = a .=. (a .+. lit1)
simplifyExpr (Assignment AssignAdd a b) = a .=. (a .+. b)
…
```

In this function we make extensive use of smart constructors such as `.+.` and `.=.`. We have specified a large number of smart constructors for various operators. These smart constructors transform expressions by simplifying them based on algebraic and logic rules and do *constant folding*, the evaluation of expressions with constant operands. We show the smart constructor for multiplication in which we first try to calculate the result if both operands are literals. If one operand is zero we return zero and if one of the operands is one (the identity element of multiplication) we return the other operand.

```
(.*.) :: Expression -> Expression -> Expression
l .*. r
    | isLit l && isLit r     = fromMaybe (l * r) (calc Multiplication l r)
    | l == lit0 || r == lit0 = lit0
    | l == lit1              = r
    | r == lit1              = l
    | otherwise              = l * r
```

We show another example for the unary boolean operator ‘not’. If possible, we eliminate the operator or push it inwards.

```
(!.) :: Expression -> Expression
(!.) (Prefixed Not e)          = e
(!.) (LitExpr (BoolLiteral b)) = LitExpr $ BoolLiteral $ not b
(!.) (Infixed Equal l r)       = Infixed NotEqual l r
(!.) (Infixed NotEqual l r)    = Infixed Equal l r
(!.) (Infixed Less l r)        = Infixed GreaterOrEqual l r
(!.) e                         = Prefixed Not e
```

For example, the expression: x += (3 + -2) will be simplified as follows:

`x += (3 + -(2))` ⇨ `x = x + (3 + -(2))` ⇨ `x = x + (3 + -2)` ⇨

`x = x + 1`

The resulting expression is similar to ‘x = 1 + x’, but this expression will not be transformed into ‘x = x + 1’ by the smart constructors. In addition we provide *views* to further standardise the format of expressions. Views can be defined using the Ideas framework and are described in Section 2.1.3. As an example, we show the view to convert an expression in the conjunctive normal form (CNF), consisting of a *match* and a *build* function. The match function `m` returns a list of conjuncts using the laws to convert to CNF and the build function recreates the expression by combining the conjuncts with the AND operator.

```
cnfView :: View Expression [Expression]
cnfView = makeView (Just . m) build
    where
        -- De Morgan 1:
        m (Prefixed Not (Infixed AND p q))  = m ((!.) p .||. (!.) q)
        -- De Morgan 2:
        m (Prefixed Not (Infixed OR p q))   = m ((!.) p) ++ m ((!.) q)
        -- Distributivity 1
        m (Infixed OR (Infixed AND p q) r ) = m (p .||. r) ++ m (q .||. r)
        -- Distributivity 2
        m (Infixed OR p (Infixed AND q r))  = m (p .||. q) ++ m (p .||. r)

        m (Infixed AND p q)                 = m p ++ m q
        m p                                 = [p]

        build xs = foldl (.&&.) trueLit xs
```

We transform an expression into CNF using this view and a function that sorts expressions. In the `sortExpr` function literals are grouped together and pushed to one side so they can be evaluated.

Other expressions such as identifiers and function calls are sorted alphabetically. Duplicate operands will be removed by the `nub` function.

```
viewAsCNF :: Expression -> Expression
viewAsCNF = simplifyWith (nub . sortExpr) cnfView . simplifyExpr
```

We apply the various views one by one in a sequence of expression transformations on programs:

```
exprTrans :: BProgram -> BProgram
exprTrans =
      transformBi viewAsCNF
    . transformBi viewAsMul
    . transformBi viewAsSum
    . transformBi viewAsUnEq
    . transformBi viewAsEq
```

As a result, we now consider the following sample expressions similar:

```
!x && !y
```

≈

```
! (y || (x && true))
```

```
a+1+f()+b[4]+4+b[3]+s()
```

≈

```
s()+b[3]+a+5+f()+b[4]
```

SPV6 Different control structures. We take the use of different control structures into account in the generation of a programming strategy, such as a for loop that can be written as a while loop and the if-then-else that can be inverted. We currently do not allow a for loop in a student solution if the model specifies a while, because it is less obvious than the other way around. However, it will not be ruled out as a future addition. We have also implemented a transformation that starts a local loop variable at zero. This transformation is only applied if the loop has a certain common format, such as in the next example:

```
for (int i = 1; i < 5; i++)
{
   f(i);
}
```

≈

```
for (int i = 0; i <= 5; i++)
{
   f(i + 1);
}
```

SPV8 Different temporary variables, SPV 9 Different redundant statements. Xu and Chee use *copy propagation,* or *forward substitution* as they call it, to standardise the use of temporary variables. For example, the following programs are considered equal and could both be written as `print("abc")`.

```
a = "a";
b = "b";
print (a+b);
```

```
c = "ab";
print (c);
```

Because we do not need variables a, b or c anymore, they can be removed and are considered *redundant statements*. Redundant statements are statements that can never be reached or statements whose result is never used (dead code). Xu and Chee apply dead code removal in their research to handle this variation.

Copy propagation and dead code removal is not a problem if it is applied to finished programs. However, some issues arise when a student submits an incomplete program. Dealing with difference in temporary variables in normalisation implies that we cannot give hints that take the students' variables into account. Let us consider some potential problems if we would implement the transformations as described, using the following example:

Model:

```
print("hello");
```

Student:

```
s = "hello";
```

To recognise the students' incomplete submission we would have to remove the (so far) unused variable assignment, which is dead code. As a result, the submission will be recognised and introducing a print statement will be the next step.

We show another, slightly different, example:

Model:

```
s = "hello";
print(s);
```

Student:

```
s = "hello";
```

If the student asks for a hint, the hint will be to introduce the variable assignment, which was already done by the student. A solution might be to not remove unused variables but mark them in some way, and relaxing the matching of two canonical programs. If the same variable was introduced in the model as well as in the student solution, we recognise it. If the variable is unknown, it will be ignored. The downside of this solution is that we would not recognise an erroneous assignment, such as `s = "goodbye"`.

Model:

```
s = "hello";
print(s);
```

Student:

```
print(?);
```

In this final example, using the same algorithm as described, the tutor would suggest replacing the hole by an identifier. This is not a correct suggestion, since the student has not created a variable yet.

Adding this variation to the tutor requires further research. It should be decided to focus on normalisation, possibly compromising on the quality of hints, or finding a way to include the use of temporary variables in the strategy while keeping the solution space and strategy size manageable. Creating strategies for every possible set of intermediate variables is an unrealistic solution. However, creating strategies for a limited set of variants is a possibility.

SPV10 Different statement orders. This variation is dealt with in the generation of a strategy.

SPV11 Different variable names. The variables in student and model programs are renamed in a normalisation. The complete program is traversed and all variables are renamed to v1, v2 et cetera. We use a state monad with a counter and a map with old and new names. Whenever we encounter a variable we have not seen before, we generate a new variable name, transform the expression and store a mapping from old name to new name in the state. If we come across a variable that is already in the map, we rename it to the new value.

SPV12 Different program logical structures. An example of this variation is that some statements can be placed either inside or outside a loop. We do not recognise these variations at the moment.

SPV13 Different statements. Xu and Chee use a ‘variation-learning process’ that is executed in a training stage to identify the final semantic similarities. Evaluation of expressions and instructor input is used to identify and store equivalent and non-equivalent component pairs. In our tutor we do not use these advanced methods, but we do allow an instructor to provide alternatives for a specific statement in an annotation without the need to create an entire new model solution, which is explained in Section 5.4.3.

5.3.2 EQUIVALENCE

When we encounter student programs in which no model solution can be recognised, we would still like to provide the student with some feedback. If we cannot recognise the inner structure, we are left with looking at the output of the program. If the output of the student program is equal to the output of a model solution, we can at least inform the student that the solution produces correct results, although we cannot comment on the algorithm used. This algorithm may either be a potential addition to the set of model solutions, or an inefficient or inelegant solution.

Testing would be an obvious tool to check if two programs produce the same output. Currently the prototype does not support functions and focuses on writing output to a console. For this reason an evaluator has been implemented that computes the output of a program based on print statements. We also have to take into account that incomplete programs may be submitted. When a student has correctly produced the first part of the output, he or she is on the right track. We therefore do not define an actual equivalence relation but instead we use a relation to define if the output of a program is a prefix of the output of a model solution. This relation is not an equivalence relation because the symmetry property does not hold. We show a simplified version of this relation in which we have omitted dealing with evaluation errors.

```
program <== sol = evaluate program `isPrefixOf` evaluate sol
```

This relation holds for the following programs that have equal output:

```
x = "*";
for(i = 0; i < 8; i++)
{
    print(x);
}
```

`<==`

```
print("********");
```

In the next example the output of the student program on the left is a prefix of the model solution on the right:

```
print("a");
```

`<==`

```
print("abc");
```

This somewhat deviating definition is related to the fact that in many other, mainly mathematical, domains student submission are expressions that should stay the same every step towards the solution. A solution that is not finished yet is most likely not the same as the final submission. In the next section we discuss how this relation is used in the diagnosis of student programs.

The evaluator itself also takes into account that the program may not be finished and may contain holes. During evaluation a state is kept containing the values of known variables, the output so far and a boolean (`complete`) indicating if no holes were encountered yet. After coming across a hole in an expression, this boolean is set to false. From there on, nothing will be written to the output anymore although all subsequent statements will still be evaluated for possible errors.

```
data EvalState = EvalState
    {
        environment :: Map VarName Literal,
        output      :: String,
        complete    :: Bool
    }
```

The type of the evaluator is as follows. The return type may be an error if the code could not be evaluated, for example when encountering a type error or a reference to an unknown value. Evaluation can be performed on any type that can be transformed into a `BProgram`.

```
evalProgram :: ToBase a => a -> Either EvalError String
```

If we would expand the prototype to support method declarations, we might be able to use (an expansion of) the evaluator for testing. An obvious choice would be to use an existing testing tool, but these tools will not support incomplete programs with holes. Therefore a custom made solution should be developed which is not within the scope of this thesis.

5.4 GENERATION OF STRATEGY-BASED FEEDBACK

Using the components described in the previous section, we can now define an exercise for the domain of imperative programming. We use the `Exercise` data type from the Ideas framework to provide our parser, pretty printer, the similarity and equivalence relation, an exercise description loaded from a text file, a unique identifier and the exercise strategy. Students can do the exercises by creating a solution and asking for feedback from the domain reasoner. We offer two feedback services that were originally designed for the Ask-Elle tutor, which we reuse for the domain of imperative programming applying some adjustments and additions:

- DEEPDIAGNOSE for diagnosing a student submission.
- ALLHINTS for providing a tree structure with hints at various levels.

Some meta services from the framework are also provided, such as loading a list of available exercises. The feedback services are described in the next sections, followed by a section on the possibilities to adapt the calculated feedback.

5.4.1 DIAGNOSIS

A student can submit a (partial) solution to a programming problem at any time. A student might even submit a finished solution straight away. After submitting, the student will receive a message indicating if the work was correct or if a mistake has been made. In the Ideas framework, a `Diagnosis` data type is available to represent the result of the diagnosis of a submitted student program, which is described in Section 2.1.2.

Because the DIAGNOSE service from the Ideas framework can only recognise single steps in a strategy, it is considered unusable in the programming domain. In programming we want to expand our program constantly while working towards the solution. In the Ask-Elle tutor the DEEPDIAGNOSE service has been added to recognise multiple steps (Gerdes, Heeren, & Jeuring, 2012). We have found that this service can be used for imperative programming as well, although we need to address some issues first.

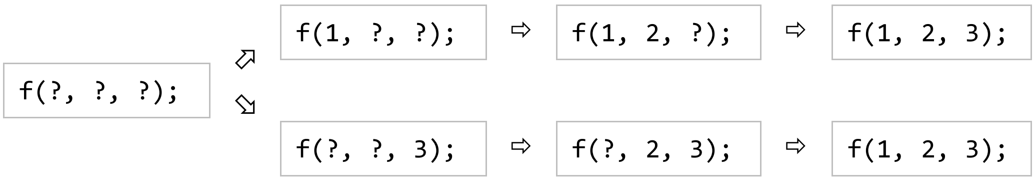
We describe the usage of the various diagnoses in Ask-Elle together with the changes for our tutor in Table 10.

DIAGNOSIS	ASK-ELLE	IMPERATIVE PROGRAMMING TUTOR
Buggy	No buggy rules have been defined.	Likewise.
Not equivalent	Unused, equivalence always returns true, although testing is implemented elsewhere.	Because an imperative program can be expanded step by step, we do not compare the previous and current submission. We determine if the output of the current program is not a prefix of the output of a model solution, in which case an error was made and this diagnosis is returned.
Similar	The current submission and the previous submission can be transformed into the same canonical form.	Likewise, although we pass an empty state instead of the previous state to the service because we allow students to remove and replace statements in a program.
Expected	The submitted program (still) follows the strategy.	Likewise.
Detour	No changes from the original diagnose service (see 2.1.2). Only the application of one rule can be recognised.	The same, but only different types of statements can be recognised because the append rule does not distinguish the different types of expression statements.
Correct	The textual diagnosis service uses QuickCheck to determine if the program passes a number of tests.	The output is a prefix of the solution but we cannot recognise what strategy the student is following.
Unknown	QuickCheck testing is unable to ascertain the correctness of the solution.	Unused.
Wrong rule	Unused.	Likewise.

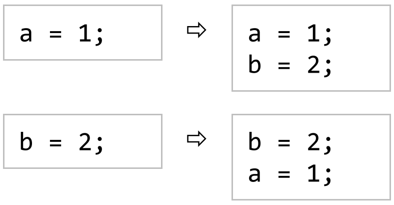
TABLE 10 DIAGNOSES FOR PROGRAMMING

The DEEPDIAGNOSE service (Gerdes, Heeren, et al., 2012) checks if the submission follows the exercise strategy by creating a list of valid prefixes. A *prefix* is an encoded list representing a sequence of rules that have already been applied. There should be at least one valid prefix that results in the program the student submitted. The list of valid prefixes is created by calculating all possible prefixes (intermediates) and keeping the ones that are similar to the student submission. To calculate these intermediates, a tree is constructed with prefixes in the nodes. This tree is created in a special search mode to reduce the size of the tree. To discard intermediates that can never be a valid prefix, the corresponding tree branches are cut. We have to address two issues if we want to use this service for imperative programming: the search mode and tree pruning.

Search mode. DEEPDIAGNOSE uses a search mode to decrease the solution space of intermediate solutions. For example, the interleave operator causes a large number of duplicate intermediate solutions, as demonstrated in these two different refinement orders:



The resulting function call can actually be reached in six different ways. If a student submits the complete call in one submission, the order in which this was done is irrelevant. DEEPDIAGNOSE reduces the search space by changing the semantics of the interleave operator so all intermediate states can only be reached by a single path. In the preceding example, in which only refinement rules are applied, this behaviour is fine and even desirable because it increases the performance of the service. In a different situation however, this causes problems:



The two statements are not dependent so they can be added to the program in any order. However, the resulting programs have a different form. The search mode recognises that rule ‘append-a’ and ‘append-b’ can be interleaved and only saves the prefix ‘append-a, append-b’, discarding prefix ‘append-b, append-a’. If a student submits the bottom program on the right in one submission, it will not be recognised by the diagnose service because the resulting program from prefix ‘append-a, append-b’ is not similar to the student submission. To prevent this from happening we only allow the deletion of paths when refinement rules are involved. All solutions will now be recognised and we still benefit from the reduction in search space for refinement rules. An alternative might be to define an insertion rule instead of an append rule specifying a specific location and not just a block. We have not explored this option as of yet, but controlling the search space for imperative programs is interesting for future research.

Tree pruning. Pruning is used to delete entire solution paths from the tree. All branches that are not predecessors of the current student submission are cut from the tree. An `isPredecessor` relation is used for tree pruning which we have defined for our `Program` data type as follows:

```
isPredecessor :: Program -> Program -> Bool
isPredecessor p1 p2 = normalise p1 ~> normalise p2
```

After normalising both programs, we determine if the first program can become the second program. The relation is both reflexive and transitive, but not symmetric. The overloaded function `~>` is defined on statements, expressions, lists and other data types. As an example, an if statement can become another if statement if the first condition can become the second condition and the first body can become the second body. An if statement cannot be the predecessor of any other statement.

```
(If e1 s1) ~> (If e2 s2) = e1 ~> e2 && s1 ~> s2
(If _ _)   ~> _          = False
```

A list of statements can be expanded with more statements:

```
[]     ~> _      = True
(x:xs) ~> (y:ys) = x ~> y && xs ~> ys
xs     ~> []     = False
```

A hole can become any expression:

```
HoleExpr ~> _ = True
```

However, there are some more issues when expressions contain holes. Let us consider a model solution to some exercise that contains the following expression:

```
a * (b + c);
```

The corresponding prefix tree represents all intermediate states, of which we show a fragment:

- `?`
  - `? * ?`
    - `a * ?`
      - `a * (? + ?)`
        - `a * (b + ?)`
          - `a * (b + c)`
          - …
    - `? * (? + ?)`
      - …

At some point a student might come up with the following incomplete expression and asks for a diagnosis:

```
? * (? + ?);
```

Applying the distribution rule, the student expression will be normalised to:

```
(? * ?) + (? * ?)
```

This expression has a different structure than ‘? * ?’ in the tree, but if we cut this node the expression that is similar to the original student submission would be lost. This issue is relevant for all kinds of arithmetic and logic expressions, which we normalise into a canonical form. Therefore, we consider all arithmetic and logical expressions that contain at least one hole to be a predecessor of any other expression. In this particular example, all intermediate states except the leaves will remain in the tree. The resulting code for the `~>` function is shown next:

```
(Call i1 args1)    ~> (Call i2 args2)    = i1 == i2
                                           && length args1 == length args2
                                           && and (zipWith (~>) args1 args2)
(ArrayAcc i1 idx1) ~> (ArrayAcc i2 idx2) = (i1, idx1) ~> (i2, idx2)
(Property i1 p1)   ~> (Property i2 p2)   = (i1, p1) ~> (i2, p2)
…
pre                ~> post
    | isStrict pre      = False -- Call, ArrayAcc, Property etc return True
    | containsHole pre  = True
    | otherwise         = pre == post
```

The relation is less flexible for some expressions, which can be seen in the first lines. For example, if the expression is a function call, the function names should be equal, as well as the number of arguments. Next we check the arguments which may be refined in any order according to the strategy. Every single argument should be a predecessor of the corresponding argument in the model. When we arrive at the arithmetic and logic expressions, we check if the predecessor contains holes. If such an expression contains at least one hole, we state that the expression can become any other expression. If the predecessor is completely refined we check if it is equal to the successor, because normalisation should have converted both expressions into the same canonical form.

5.4.2 HINTS

To provide textual hint messages *labels* are used to annotate rules and sub strategies. During the generation of a strategy for an exercise, labels are attached to certain parts of the strategy. For imperative programs, the following labels and other descriptions are inserted automatically:

- Rules have an identifier with a description, for example ‘Introduce break-statement’ for an append rule and ‘Expand ? to identifier’ for a refine rule.
- Sub strategies are labelled to provide more specific feedback. We provide an example for a for statement, that has the following format:
  `for (init; cond; incr) body;`
  During the generation of a strategy, the sub strategy for init will be labelled with ‘loop-init’, the sub strategy for cond with label ‘loop-condition’, the sub strategy for incr with ‘loop-incr’ and the sub strategy for body with ‘loop-body’. We show a fragment of the strategy generation for a for statement in which we return a labelled strategy.

```
return $ appendFor
    <*> label "loop-init" init'
    <*> (     label "loop-condition" (atomic $ sequenceS cond')
          <%> label "loop-incr" (atomic $ sequenceS incr') )
    <*> label "loop-body" body'
```

The corresponding textual descriptions for labels are stored in a text file. The Ideas framework supports the parsing of these files into a *feedback script*. This script is used to generate textual feedback messages. The Ideas framework provides a `Script` data type that stores textual representations for several components such as rules, strategies and diagnoses. The input for the script used in Java exercises is loaded from a text file that can be manually adjusted by an instructor. We show a small fragment from this feedback text file:

```
feedback loop-incr= What to do after each loop iteration?
feedback assign   = What value should the variable get?
feedback args     = What information should you pass to the function?
```

The ALLHINTS service from the Ask-Elle tutor is used in our tutor to generate a tree structure with hints on how to proceed, as shown in the following example:

- `Introduce a loop statement`
    - `Introduce a for statement.`
        - `Type code for (?; ?; ?) {}`
    - `Initialise a variable for a while statement`
        - `Expand ? to a variable assignment.`
            - `Type code i = ?;`

The hints are based on the steps defined in the exercise strategy and the corresponding labels. The branching indicates the choice between different steps and the depth of a node indicates the level of detail of the feedback message. The ALLHINTS service in its turn uses the ALLFIRSTS service which is a rewrite for Ask-Elle of the original service from the framework. The adjusted service works with a list of prefixes to avoid the issue of model solutions sharing their first step, as described in Section 2.2.4. Both services are slightly adjusted because we currently do not support the use of a 'name map'. A name map stores a mapping between identifiers used by the student, such as variable names, and the corresponding identifiers from the normalised student program.

In this section and the previous section we have shown that the services developed for the functional programming tutor Ask-Elle can also be used for imperative programming. Moreover, the services are generic, in the sense that they support doing multiple steps at once, regardless of the nature of the steps. We propose moving the services to the Ideas framework to make them more widely available. However, some additions and alterations to the services are necessary. Some functions used by the services, such as `isPredecessor`, should be made generic by leaving the implementation to the domain that uses them. We should note that this relation might not be useful to domains in which doing multiple steps at once is not desirable. Additional features such as name mapping could be made optional. Furthermore, the code for some diagnoses of the DEEPDIAGNOSE service is equal to the standard DIAGNOSE service. We could investigate if and how a diagnose service can be dynamically composed of functions for the different `Diagnosis` types.

5.4.3 ADAPTING FEEDBACK

If an instructor wants to use our tutor for a particular exercise, the instructor only needs to provide a set of model solutions. Feedback will be calculated automatically based on these solutions. However, an instructor may sometimes want to provide additional information to further guide the process of solving an exercise. A number of instructor facilities are implemented in the prototype. The script that stores the textual representations for strategy labels and other feedback messages can easily be adjusted by an instructor. The model solution that the instructor provides can be customised with several annotations. These annotations enable instructors to create tailor-made exercises for their students.

The Ask-Elle tutor introduced the concept of annotated instructor solutions (Gerdes, Heeren, et al., 2012). We have adopted a number of these annotations in our tutor for imperative programming. We also propose some adjustments. Annotations can be added to the model code inside comments so they do not cause compiler problems. The Java parser is able to recognise these comments. The `Program` data type has been adapted to accommodate the annotations. We will describe the features that are currently available in the prototype.

General solution information. A model solution can be annotated with general information. At the top of the model solution the following annotation can be added:

```
/* DESC "Implement the Quicksort algorithm" PREF 2 DIFF Hard */
```

This information can be stored in a `Program`:

```
data Program = Program
    {
        body       :: Statement,
        desc       :: String,
        difficulty :: Difficulty,
        preference :: Int
    }
```

We label the strategy for a particular model program with the solution description. The solution difficulty is currently unused. When we take the student level into account we might exclude certain solution paths because they are either too difficult or too easy. We use the preference number in rule ordering to show the hints that lead to the most preferred solution path first.

Feedback messages. The FEEDBACK-annotation can be used to provide more information about the semantic meaning of a statement in the context of a specific assignment, for example:

```
/* FEEDBACK Calculate the average of the two results */
double avg = (x + y) / 2;
```

Another example is:

```
/* FEEDBACK Create a loop through all even numbers below 100 */
for (i = 0; i < 100; i += 2) … ;
```

The feedback text will be attached to the statement that follows the annotation. The `Statement` data type has been extended in the following way:

```
data Statement = … | Feedback String Statement
```

When we derive a strategy from a statement that is annotated with a feedback message, we create a new unique identifier for this label and attach it with the text as a description to the corresponding strategy for the statement itself.

```
genStrat loc pref (Feedback msg stat) ->
    liftM2 attachFb getNextNr (genStrat loc pref stat)
        where
            makeLabel   = describe msg . newId . ("fb." ++) . show
            attachFb id = toStrategy . label (makeLabel id)
```

Mandatory language constructs. Occasionally an instructor may devise an exercise to train using a particular new language construct. For example, when introducing the for statement, the students should practise with this statement and not revert to a while statement that they might already know. Because the strategy generator by default attempts to include as many variants as possible into the strategy, the generator should be instructed when this is not desirable. The instructor is able to do this by annotating a statement in a model solution using the MUSTUSE-annotation.

```
/* MUSTUSE */ for (int i = 1; i <= 10; i++) …;
```

The `Statement` data type has been extended in the following way:

```
data Statement = … | MustUse Statement
```

This annotation will instruct the strategy generator not to include the option to create a while statement as an alternative. In Ask-Elle this annotation is used to prohibit the recognition of the underlying implementation of a library function instead of the library function itself. In the imperative tutor this annotation is currently used to enforce the use of a for or while statement and can be used for other language constructs in the future.

Alternatives. In some cases we want to allow an alternative for a single statement. Creating an entire new model solution for this is too much work and does not make sense for just one line of code. Using the `ALT`-annotation, we can provide an alternative for one specific statement that follows the annotation. As an example, we use the annotation to allow a library function instead of one's own implementation. Note that we currently restrict the implementation to one statement, but could expand this to multiple statements in the future.

```
/* ALT x = Math.max(a,b); */
if (a > b)
    x = a;
else
    x = b;
```

The `Statement` data type has been extended with an `Alt` constructor that accepts a list of statements of which either one is allowed.

```
data Statement = … | Alt [Statement]
```

If we encounter this constructor while deriving a strategy, we will generate a strategy for each statement and combine them using the `alternatives` combinator.

```
genStrat loc pref (Alt stats) =
    liftM alternatives $ mapM (genStrat loc pref) stats
```

Feedback level. When generating a strategy for a set of models, a level is passed as a parameter to indicate the granularity of the steps in the strategy. For example, a particular exercise may be targeted at more advanced students who do not need feedback at a very low level. As a default, all strategies are generated at the lowest level of one. Strategies for level two do not include refinement rules that help developing a composed statement step by step. Not including refinement rules implies that a student can no longer use the hole (?) symbol as a placeholder for unknown expressions.

Let us consider the following model program as an example:

```
if (x > 10) f();
```

The feedback level can be set in a configuration file in the exercise folder. If the level is set to one, the first hints are:

- `Introduce an if statement.`
    - `Type code if (?) {}`

If the level is set to two, the hints are:

- `Introduce an if-statement with condition: an expression with operator >.`
    - `Type code if (x > 10) {}`

In the code for the `genStratWithLoc` function we generate a new hole and a corresponding strategy for an expression if the level is one. For level two (other levels are currently not supported) we simply return the complete expression paired with the `succeed` strategy from the framework that always succeeds, to prevent adding refinement steps to the resulting strategy.

```
genStratWithLoc pref expr = getLevel >>= makeHoleAndEx
    where
        makeHoleAndEx level
            | level == 1 = do
                loc   <- getNextNr
                expr' <- genStrat loc pref expr
                return (HoleExpr loc, expr')
            | otherwise  = return (expr, succeed)
```

A possible expansion is to omit adding certain labels so they will not appear in the hint tree.

6 VALIDATION

We have proposed a number of questions in Section 4.3 to validate the results of this research, which we repeat here for convenience:

a. Are student programs that do or do not follow a known strategy recognised as such?
b. Can a step (or multiple steps) in a student program be recognised as either following a known strategy or not?
c. Do the generated hints lead the student to a solution?
d. Does the generated feedback reflect the annotations in model solutions?

We collected data from first year IT-students from Windesheim University of Applied Sciences during their Web programming course from September to November 2013 and their Java programming course from February to April 2014. The students are both full-time and part-time students and are enrolled in one of the four IT-studies of Windesheim: ‘Software Engineering’, ‘Business IT and Management’, ‘Infrastructure Design and Security’ and ‘Embedded Software and Automation’. We asked the students to solve a number of programming problems and submit their (either complete or incomplete) solutions. We have used these solutions for the validation of our research.

In this chapter we answer the validation questions and describe how we arrived at the conclusions. Our test suite is described in Section 6.1. In Section 6.2 we show two tutoring sessions that give an impression of the behaviour and capability of the prototype. This demonstration provides answers for validation questions b, c and d. In Section 6.3 we describe the analysis of a large number of sample programs created by the students, contributing to question a. In this analysis we study the large solution space and show the variations our tutor supports. We conclude in Section 6.4 with summarizing the answers to the validation questions.

6.1 TESTING

We have created a test suite to automate the testing of various cases. Test data was both invented and generated randomly (see Section 5.1). Our test suite includes:

- Tests for the normalisations (Section 5.3.1).
- Tests for recognising variants that are included in the generation of a strategy (Section 5.2.2).
- Tests for instructor annotations (Section 5.4.3).
- Tests for the evaluator (Section 5.3.2).
- Tests for recognising steps with different granularity.
- Tests for recognising steps in different orders.
- Simple performance tests for dealing with larger programs.
- Limited tests that check the number of expected hints.

6.2 TUTORING SESSIONS

6.2.1 PHP TUTORING SESSION

Our first session revolves around a small and simple PHP exercise with the following description:

‘Write code to show the following string on the screen:

*+**+***+****+*****+******+*******+********+*********+**********+’

The exercise was included in the set of web programming exercises for which we collected data. We provide two model solutions. The model solutions are not annotated because we currently do not support annotations in PHP. The first model uses two nested loops:

```
<?php
    for($i = 1; $i <= 10; $i++)
    {
        for($j = 1; $j <= $i; $j++)
        {
            print("*");
        }
        print("+");
    }
?>
```

The second model solution uses one loop and a library function:

```
<?php
    for ($i = 1; $i <= 10; $i++)
    {
        print str_repeat("*", $i) . "+";
    }
?>
```

For this session we have used the command line interface to our tutor. The solution we show in the session is selected from the data set of student solutions. Because we do not know how the student arrived at this solution, we simulate a path the student could have taken. After each student submission we show the diagnosis of the tutor together with the tree of hints. The leaves always show the program code that should be typed and will be omitted later in the demonstration.

Student The student does not know how to start and asks for a hint.

Tutor Both a for and a while statement are suggested:

```
Hints:
•   Introduce a loop statement
        •   Introduce a for statement.
                •   Type code for (?; ?; ?) {}
        •   Initialise a variable for a while statement
                •   Introduce a variable assignment.
                        •   Type code i = ?;
```

Student The student chooses to implement a for statement:

```
for (?;?;?) { }
```

Tutor

```
Correct
Hints:
•   What to do before looping?
        •   Expand ? to a variable assignment.
```

Student The student initialises a loop variable in one step.

```
for ($i = 1;?;?) { }
```

Tutor

Correct

Expanding both parts of the for loop is now permitted:

Hints:

- When to continue looping?
    - Expand the second part of the for-statement
        - Expand ? to an expression with operator <=.
- What to do after each loop iteration?
    - Expand the third part of the for-statement
        - Expand ? to an expression with postfix operator ++.

Student

The student tries to complete the loop condition, but makes a mistake.

```
for ($i = 1; $i < 10;?) { }
```

Tutor

We lost you

Student

The student fixes the mistake.

```
for ($i = 1; $i <= 10; $i++) { }
```

Tutor

We can recognise both model solutions in the hints:

Correct.

Hints:

- What to repeat?
    - Introduce a for-statement.
    - Initialise a variable for a while statement
        - Introduce a variable assignment.
    - Introduce a print-statement.

Student

The student chooses the third option and already knows that two components should be concatenated and printed.

```
for($i = 1; $i <= 10; $i++) {
    print ? . ?;
}
```

Tutor

Correct.

Hints:

- What to repeat?
    - What do you want to print?
        - Expand ? to a method call to str_repeat with 2 params.
        - Expand ? to a literal "+".

Student

The student refines both holes.

```
for($i = 1; $i <= 10; $i++) {
    print str_repeat(?, ?) . '+';
}
```

Tutor

```
Correct
Hints:
•  What to repeat?
       •  What do you want to print?
              •  What information should you pass to the function?
                     •  Replace the ? in the 1st argument
                              •  Expand ? to a literal "*".
                     •  Replace the ? in the 2nd argument
                              •  Expand ? to an variable.
```

Student The student finishes the program.

```
for($i = 1; $i <= 10; $i++) {
    print str_repeat('*', $i) . '+';
}
```

Tutor

```
Correct
You are done!
```

6.2.2 JAVA TUTORING SESSION

We continue with another exercise that was part of the set of Java programming exercises for which data was collected. The exercise has the following description:

‘Create an array with the following sequence of numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

We are going to check if this is the correct sequence of Fibonacci numbers. The Fibonacci sequence starts with 1, 1, and every next number should be the sum of the two previous numbers.

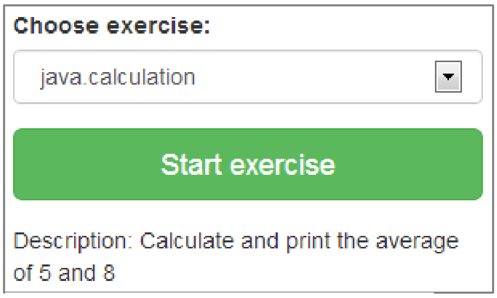
Write code to perform this check. You do not have to check the first two numbers. Print the word 'correct' or 'incorrect' on the screen to check the result.’

We initially created one annotated model solution:

```
/* DESC "Use a loop to check the numbers in an array" PREF 1 DIFF Easy */

/* FEEDBACK Use the {}-syntax to put the fibonacci numbers in an array */
int [] fib = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
/* FEEDBACK Declare a boolean to store the result of the checks */
boolean isCorrect = true;

/* FEEDBACK Create a loop with a counter starting at 2 and ending at the end of
   the array */
for (int i = 2; i < fib.length; i++)
{
      /* FEEDBACK Check if the number at the index is not the sum of the numbers
          at the previous two indices */
      if (fib[i] != fib[i-1] + fib[i-2])
      {
                 isCorrect = false;
      }
}
/* FEEDBACK Check the boolean value and print the right text accordingly */
```

```
if (isCorrect)
{
	System.out.println("correct");
}
else
{
	System.out.println("incorrect");
}
```

For this session we use the web interface. In the web interface the student can select an exercise from a list of available exercises (Figure 18). Figure 19 shows a screenshot of the editor that has syntax highlighting where the solution can be typed. There are buttons that generate templates for some statements on the top and buttons to ask for feedback on the bottom.



java.calculation

Description: Calculate and print the average of 5 and 8

FIGURE 18 WEB EXERCISE SELECTION

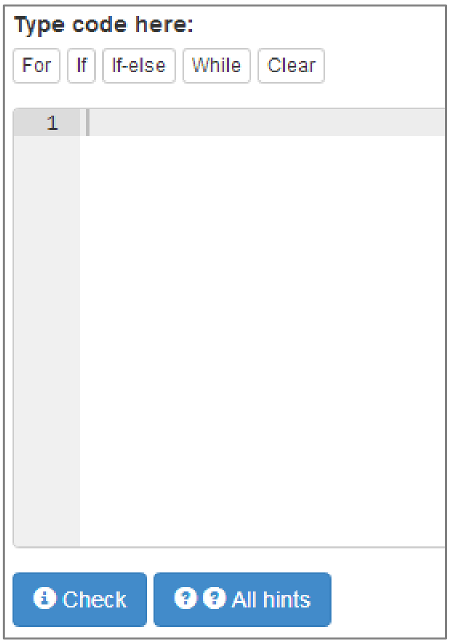
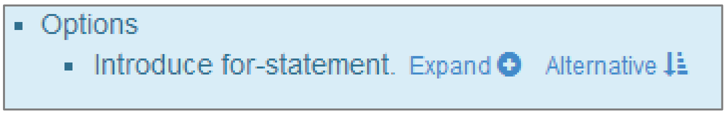


FIGURE 19 WEB EDITOR AND FEEDBACK BUTTONS

In the web interface we show the first option (branch) of the tree when the student first asks for help (Figure 20). The student has the opportunity to ‘expand’ on a specific path and view a hint with more detail. If other options are allowed, the ‘alternative‘ link will provide a hint on a different solution path. In this session we show screenshots of the hint tree, sometimes omitting certain hints such as showing the actual code that should be typed.

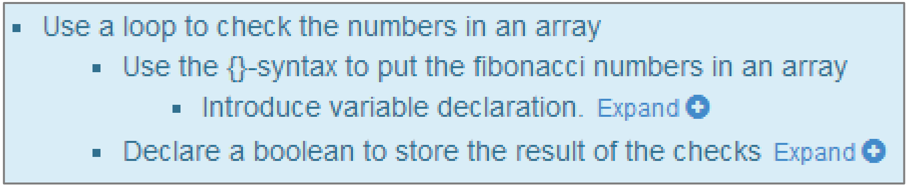


- Options
  - Introduce for-statement.

FIGURE 20 HINTS IN THE WEB INTERFACE

Student The student does not know how to start and asks for a hint.

Tutor The solution description of the model is always shown at the top of the hint tree.



- Use a loop to check the numbers in an array
    - Use the {}-syntax to put the fibonacci numbers in an array
        - Introduce variable declaration. Expand
    - Declare a boolean to store the result of the checks Expand

Student The student uses a different syntax and a different variable name for the array declaration.

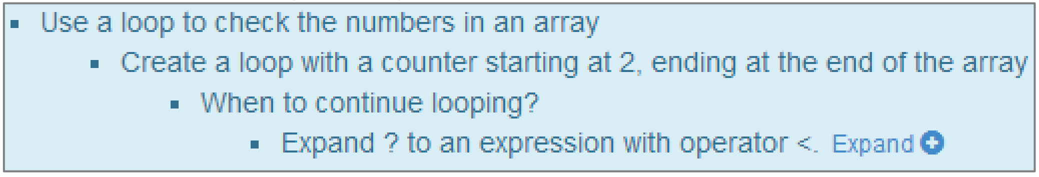
```
int nrs[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
```

Tutor Correct.

Student The student continues programming, starts a while loop and asks for a hint.

```
int nrs[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
boolean isCorrect = true;

int i = 2;
while(?) {

}
```

Tutor The start of the while loop is recognised.

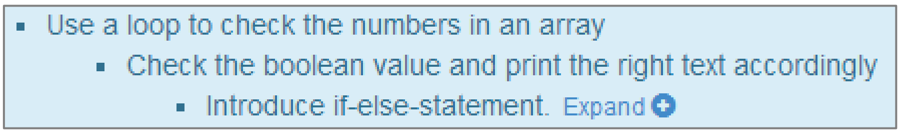


- Use a loop to check the numbers in an array
    - Create a loop with a counter starting at 2, ending at the end of the array
        - When to continue looping?
            - Expand ? to an expression with operator <. Expand

Student The expression in the if is slightly different from the model solution.

```
…
while(i < nrs.length) {

    if (nrs[i] != nrs[i-2] + nrs[i-1]) {
        isCorrect = false;
    }
    i++;
}
```

Tutor Correct.



- Use a loop to check the numbers in an array
    - Check the boolean value and print the right text accordingly
        - Introduce if-else-statement. Expand

Student The student switches the true and false branches of the if statement.

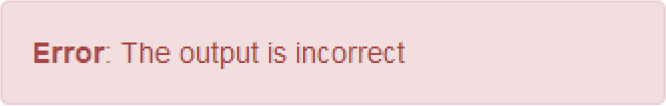
```
…
if (!isCorrect)
    System.out.println("incorrect");
else ;
```

Tutor Correct.

Student The student makes a mistake.

```
…
if (!isCorrect)
    System.out.println("incorrect");
else
    System.out.println("incorrect");
```

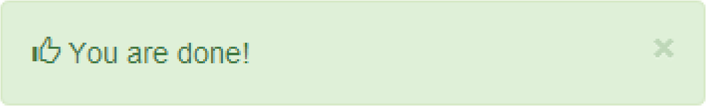
Tutor



Error: The output is incorrect

Student The student fixes the mistake and has completed the exercise.

Tutor



👍 You are done!

6.3 ANALYSING STUDENT PROGRAMS

The set of collected student programs provides us with information about different solutions. Although we do not know how an individual student arrived at his or her solution, it is still relevant to analyse the submissions to find out the diversity and to determine to what extent our tutor can handle this diversity. Because it is not our intention to recognise as many variants of a program as possible, we look at how many programs we can recognise that we actually *want* to recognise in our tutor because they closely match the instructor's solution. This is a different approach from most assessment tools. The results give us an indication of the capabilities of the tutor and provide information on the variations that actually occur in student programs that we have not taken into account yet.

We collected student solutions for six different exercises, four of which are exercises from a web programming course in PHP and two are exercises from a Java course. We have found that not all of these exercises are currently suitable for validating our tutor for two reasons:

- Input can be chosen by the student by assigning a random value to a variable. As a result, our tutor would consider all solutions to be different.
- The exercise requires the student to write a function definition, which is not supported by the tutor.

For our analysis we only take the exercises we consider suitable into account.

6.3.1 PHP EXERCISE ANALYSIS

For two PHP exercises we checked the solutions using our tutor. The PHP exercises are relatively simple; their solution consists of few lines of code containing basic constructs such as loops, variable assignments and conditional statements. The results can be viewed in Table 11. Our tutor is capable of recognising 75% (for the first exercise) and 33% (for the second exercise) of the solutions that we consider similar to a model if they would be manually assessed. Unfortunately there were very few students with a decent solution for the second exercise. No false positives were identified. There were even more correct solutions, but they used different algorithms for which we should have added a model solution.

CHECK	EXERCISE 1 'STARS'	EXERCISE 2 'SUM'	EXPLANATION
Submitted solutions	60	49	
Model solutions	2	2	
Tutor parser	54	44	A small number of programs contain syntax errors. We do not include these solutions. Most programs that could not be parsed contain language constructs that are not yet supported by the prototype, such as function definitions and enhanced for loops.
Recognised as similar to model by tutor	24	2	More precisely, we mean similarity to a program that is the result of following the strategy for the exercise.
Similar to model by hand	32	6	The submitted solutions were manually assessed. A solution was put in this category if it was close to a model solution.
Correctly recognised	75%	33%	This (rounded) percentage represents how many solutions were actually recognised by the tutor in comparison to the number the instructor marked as similar to model.

TABLE 11 PHP STUDENT PROGRAM ANALYSIS

6.3.2 JAVA EXERCISE ANALYSIS

In one of the tutoring sessions we used the Java exercise in which an array should be checked if it contains the valid Fibonacci sequence. This exercise is more complex and has a larger solution than the PHP exercises. We have used this exercise to assess the possibilities and limitations of the tutor in its current form, but also to analyse the diversity in student solutions. Looking at the submissions, it was clear that the students had quite some difficulties solving this exercise with very limited to no help from an instructor.

After analysing a large number of student solutions it became clear that many students had understood the exercise differently than it had been intended. Instead of printing the correctness of the entire sequence, they printed the result of every single number check. Therefore we decided to include another (suboptimal) model solution to extend our solution space:

```
int [] fib = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};

for(int i = 2; i < fib.length; i++)
{
    if (fib[i] == fib[i-1] + fib[i-2])
    {
        System.out.println("correct");
    }
    else
    {
        System.out.println("incorrect");
    }
}
```

In the model solutions the ALT-annotation is used to allow for similar output, such as ‘Correct’ starting with a capital. As a result, one exercise can have different output, which in itself is not desirable. However, for this experiment it provides us with more information on the results without having to dismiss several student solutions solely because they did not precisely follow the instruction.

In this analysis we are using our tutor for the assessment of student solutions to programming exercises, which is a different but related area of research that we explored in Section 3.4. We are confronted with some of the issues related to automatic assessment, which are described by Pieterse (Pieterse, 2013) among others. In this publication ‘clear formulation of tasks’ is mentioned as a success factor, meaning that being extremely precise about the requirements is a necessity. The author uses regular expressions to allow more flexibility with the output of a program. In our tutor we consider different output as an error: we want to make the student aware of the importance of carefully reading the requirements of an exercise.

We examined all student solutions using our tutor and a Java compiler, and determined all variations compared to the model programs by hand. Because the Java parser does not deal with classes and methods, we have manually stripped the student solutions from all code (package, imports, class, main method header) except for the body of the main-method. Two student solutions were discarded because a function was written instead of putting the code in the main function as instructed.

General analysis. We show an overview of the submissions of 80 students in Table 12. In our detailed analysis we exclude programs that contain a Java compiler error. We consider all remaining 72 programs to identify general minor variations by hand in Table 14. We include the program in our count if the variation occurs at least once in the program. We sort the variations by the number of occurrences. For every variation we show if our tutor is able to deal with it using the symbols from Table 13. We also link the variations we identified to the semantics-preserving variations (SPV's) we discussed in Section 5.3.1 to see which of them actually occur in Java programs from students and to identify variations we might have missed.

CHECK	# PROGRAMS OK	# PROGRAMS WITH ERROR	CAUSES
Java compiler	72	8	Two students gave up after a few statements and added random text or a final unfinished attempt. One student created a seemingly complete solution but with an invalid expression.
Java evaluation	68	4	All runtime errors are 'array index out of bounds' exceptions.
Tutor parser	52	28	The tutor does not support: • Enhanced for loops • Integer and Boolean object type, long type • Conditional ?-operator The remaining errors were also detected by the Java compiler.
Tutor evaluation	48	4	Two evaluation errors are tutor issues: • One student used a call to System.exit(0), which is not supported. • One student initialised a string to null, which is not supported. The other two errors are runtime errors for an array index out of bounds that also occur in the Java runtime.

TABLE 12 OVERVIEW

TUTOR SUPPORT	SYMBOL
Supported	✔
Should be supported	+
Not supported	✕

TABLE 13 SYMBOLS

SPV	VARIATION	#	SUPPORT	EXPLANATION
11	Different variable names	72	✓	Every student uses at least one different name for a variable.
3	Filling the array element by element	24	✓	Every assignment is on a different line.
3	Different array declaration syntax	9	✓	Some students implement the variations mentioned under SPV3 in Section 5.3.1.
13	Hard-coded number of loop iterations	6	+	It is preferred to refer to the array length so this option will not be included in the strategy. However, we do want a transformation that replaces the length property by an actual value if it is known.
6	Using a while instead of a for	3	✓ / +	A while is only recognised if the loop counter initialisation and increment are at the right locations, which is the case in one solution. We would like to make this more flexible in our tutor.
13	Different data types	2	✕	The Integer and Boolean object type and the long type are currently not recognised and will not be supported as a variation either. There is no need to use the object types or the long type if the model does not use them.
7	Expressions that can be simplified	2	✓	Students use ‘… == false’ or ‘… == true’.

TABLE 14 GENERAL VARIATIONS

Detailed analysis. We have categorised the solutions by hand according to the algorithm used, as shown in Table 15. Alternative 1 was implemented by a small number of students. We had not foreseen this new solution, for which we could include a new model. The solutions in the other categories should not be recognised. Alternative 2 is not the intention of the exercise and alternative 3 is an inefficient solution. We continue with the remaining 57 solutions that at first glance attempt to solve the problem according to one of our model solutions. We have analysed the programs by hand searching for errors and variations. The errors we have found are summarised in Table 16. We have identified the variations with respect to our models in Table 17, of which some can be identified as unwanted and others as permitted alternatives.

ALGORITHM	#	DESCRIPTION
Model solution 1	23	Checking the numbers in a loop and printing the end result.
Model solution 2 (suboptimal)	34	Checking the numbers in a loop and printing the result of every single check.
Alternative 1	4	Checking the numbers in a loop and counting the number of errors, followed by a print of the result.
Alternative 2	2	Recreating the Fibonacci sequence.
Alternative 3	1	Using a large if statement to check the numbers.
None	8	The program is not a serious attempt or we cannot make sense of the code.

TABLE 15 CATEGORISATION BY ALGORITHM

ERROR	#	EXPLANATION
Different output format	22	Students use many variations such as ‘Goed!’, ‘Success’, ‘De fibonacci reeks klopt’ (the Fibonacci sequence is correct) but also ‘1+1=2Goed..’.
Incorrect number of iterations in loop	18	Some students include the first two numbers that did not need to be checked, some forget the first or last number and some cause a runtime error by trying to check numbers outside the array bounds.
Incorrect check if two subsequent numbers add up to the next.	11	Some students use new variables instead of checking the actual values from the array. Other students simply implement an incorrect addition or do not add up anything at all.
Different conditional structure	4	Incorrect variants are: an if without an else, a larger if-else structure, an if-then-else in which the true and false branches are swapped mistakenly.
Other	5	Nested loops, missing output, incorrect array.

TABLE 16 ERRORS

SPV	Variation	#	Support	Explanation
8	Extra variables for intermediate storage	28	✚	Some extra variables are only used for storing a previous index or number and should be recognised.
5,7	Different expression format for check if two subsequent numbers add up to the next	22	✓	Different operand orders.
12	If-else instead of if statement	11	✗	In all cases the solution was made unnecessarily complex.
5, 7	Different expression format for check if two subsequent numbers do *not* add up to the next	11	✓	Operands are in a different order or different operators are used.
6	Different loop counters	15	✓ / ✚	There are multiple options to check elements 2 – 11 in the array. Variants use the index of the middle element or the first element as the loop counter. We have included this in a normalisation for a limited number of cases.
12	Different number of loop iterations with a bounds check inside the loop	9	✗	This option can be prevented by using a while or for loop with the right counter and will not be supported.
1	Using a break to exit from loop	8	✓	A break statement can be used to quit checking numbers once an error has been identified. It changes the algorithm so we need a new model, but we could also use the Alt-annotation to use a break as an alternative to an empty statement.
1	Using a string instead of a boolean to store the result of the check	8	✓	Because this solution changes the algorithm so much, we propose adding a new model solution.
4	The variables from the models are declared and initialised separately	7	✓	
8	Unnecessary extra variables	5	✗	Some students use counter variables for all three elements in the calculation, which we consider an inefficient solution. Other students use other unnecessary extra variables that make their solution more complex.
6	?-operator instead of if statement to print final result	3	✚	The use of this operator should be normalised into the same form as the use of a similar if statement.
12	Enhanced for loop with extra check	3	✗	Because using an enhanced for does not make sense in this algorithm, the option will not be supported. Note that there are other cases in which an enhanced for does offer a suitable alternative.

10	Difference in order of known statements	3	✔	In some solutions the order of the boolean and integer array declarations were switched.
6	Different if-else structure	1	✔	The check is negated and the true and false branches are switched.
1	No intermediate variable	1	✔	Not using a boolean or string but printing the result immediately when possible, combining it with a return statement. This solution requires an extra model.
var.	Other	8	✖	Unnecessary checks, unused variables, strange language constructs et cetera.

TABLE 17 VARIATIONS TO MODELS

Results. Summarising the results we have identified the number of programs that our tutor should recognise in Table 18. Currently the number of recognised solutions is lower because not all variants have been implemented in the prototype and we do not support all language constructs. Many programs contain multiple variations, which explains why the percentage is not that high: with only one unrecognised variation the entire program is discarded. We have also discovered that more model solutions are needed to support some variations that deviate too much from our models.

	EXERCISE 3 'FIBONACCI'	EXPLANATION
Total number of solutions	57	
Model solutions	2	
Recognised as similar to model by tutor	5	
Similar to model by hand	12	We count the programs that only contain permitted variations. We do not count the programs for which we need a new model.
Correctly recognised	42%	This (rounded) percentage represents how many solutions were actually recognised by the tutor in comparison to the number the instructor marked as similar to model.

TABLE 18 SUMMARY

Note that some of the SPV's we have identified and do not want to include in our tutor are variations number 1 (different algorithms) and number 13 (different statements). It is still the decision of an instructor to add a new model solution or use the ALT-annotation to allow alternative statements when all other transformations are not sufficient to recognise a program. However, we would like to use our tutor to only guide students towards elegant solutions, thereby limiting the solution space.

6.4 CONCLUSION

We conclude this chapter by revisiting the validation questions and providing answers based on the findings from this chapter.

a. Are student programs that do or do not follow a known strategy recognised as such?

We have found that we can recognise 33%, 42% and 75% of the solutions that are similar to a model (for three different exercises). We have shown in our analysis that our tutor can handle a number of variations that occur in actual student solutions, although we are by no means complete. Additional variations have been identified in our analysis. We have also created a test suite with many test cases that test the diagnosis of solutions with several variations.

b. Can a step (or multiple steps) in a student program be recognised as either following a known strategy or not?

We have shown two scenarios that illustrate how multiple steps are correctly diagnosed by the tutor. Our test suite also incorporates testing various sequences of submissions.

c. Do the generated hints lead the student to a solution?

In our scenarios we have shown that following up on the hints of the tutor leads us to a correct solution.

d. Does the generated feedback reflect the annotations in model solutions?

In the scenario we can see the various FEEDBACK-annotations reappearing in the feedback, as well as the exercise description and the level. The analysis includes the ALT-annotation. Our test suite shows that the MUSTUSE- and ALT-annotations are recognised.

7 CONCLUSION

We conclude this thesis by summarising the answers to the research questions from Section 4.1. We list the contributions our research has made to both the field of intelligent tutoring for imperative programming and the research on using the Ideas framework for programming. We discuss the relation of our work to other tutors and suggest a number of areas for further research.

7.1 ANSWERS TO RESEARCH QUESTIONS

i. What are the differences and similarities between the domain of imperative programming and the domains that have already been implemented using the Ideas framework?

The domain of programming is different from the mathematical domains because other rules than solely rewrite rules are used and doing multiple steps at once is allowed. Compared to the functional programming domain, we expand a program instead of only refining it. For both paradigms we do allow multiple steps. Expanding programs also has consequences for the equivalence of terms, which is only relevant for a finished program. Instead, we define an antisymmetric relation 'isPrefixOf' to compare the output of imperative programs.

ii. How do we construct a strategy for solving an imperative programming exercise?

We have designed an abstract syntax for simple imperative programs that includes a selection of basic imperative language constructs, such as loops, branching statements and variable assignments. We have developed a strategy generator that derives a programming strategy from a set of model solutions. The strategy describes the steps to arrive at one of these models. Steps can expand a program with new statements or gradually refine a template for a particular statement. We incorporate alternative paths in the strategy for both the order of the steps and some allowed variants of language constructs.

iii. How do we represent incomplete imperative code?

Imperative programs can be incomplete by simply omitting statements. Expressions used in statements, such as variable assignments and loop conditions, can be represented with a question mark (hole), indicating that the exact implementation is unknown.

iv. How do we distinguish different solutions to an imperative programming problem and when are solutions similar?

Besides incorporating alternatives in the strategy, we perform a number of transformations on programs. These transformations are a subset of the semantics-preserving variations identified by Xu and Chee (Xu & Chee, 2003). In a tutoring situation we want to limit the number of variations because some variations may be unnecessary or unwanted. Moreover, recognising solutions that deviate too much from a model solution may result in feedback that is less accurate.

v. How can we recognise a strategy in (incomplete) imperative code?

We use the Ask-Elle DEEPDIAGNOSE service for diagnosing a student submission. We have made some adjustments and additions to enable the service for the imperative programming domain, such as the implementation of a predecessor relation that establishes if a program can be expanded to another program. An evaluator has been created to inspect the output of a program if no strategy can be found.

vi. How can we generate semantic feedback for (incomplete) imperative programs?

Labels are attached to rules and sub strategies during the generation of a strategy for an exercise. The labels have got corresponding textual hint messages that will be used for feedback. The ALLHINTS service from the Ask-Elle tutor is used to create a tree of feedback messages.

vii. How can feedback be adapted by an instructor?

We have implemented facilities for instructors to annotate a model solution to further control the feedback. The instructor can provide more specific semantic feedback for a statement, enforce the use of a particular language construct, allow alternative statements, control the granularity of the steps and define the preferred order of the models.

7.2 CONTRIBUTIONS

We have created a prototype for a tutor that generates feedback for the stepwise solving of imperative programming exercises. In Table 19 we show an overview of the features of our prototype just as we did for related work on programing tutors in the literature study in Chapter 3

	TUTOR FOR IMPERATIVE PROGRAMMING
Main paradigm	Imperative
Problem classification	Class 2 and 3
Feedback and hints	✓
Handles partial programs	✓
Handles multiple steps	✓
Knowledge base	Problem description and annotated model solutions
Handling of syntax variation	✓ , to a certain degree
Handling of algorithm variation	✓
How to assess program correctness	Strategy derived from annotated model solutions

TABLE 19 TUTOR FEATURES

We have made a number of contributions to the field of intelligent tutoring for imperative programming:

- Our tutor supports class 3 problems in the categorisation by Le et al. (N.-T. Le et al., 2013): students can practise with imperative programming exercises that can be solved by multiple algorithms.
- Our tutor accepts incomplete programs and focuses on the step by step development of a solution. The feedback indicates if the student is on the right track and gives hints on the various ways on how to proceed.
- We use a general purpose strategy language to define the process of creating a solution for an imperative programming exercise. We derive strategies for the creation of a number of common language constructs in imperative programming, such as for and while loops, if-then-else statements, variable declarations and variable assignments. For a list of statements (the program itself and nested statement blocks) we have introduced a new strategy combinator that lists all possible sequences taking dependencies between statements into account. We combine multiple ways to build up a program, which at the moment are expanding the program line-by-line and refining templates for certain statements.

- We deal with variation partly by encoding different options into the strategy and partly by performing a number of transformations.
- Feedback is based on an exercise description together with a set of annotated model solutions provided by an instructor. A model solution can be annotated with a description, order of preference, specific feedback messages, alternative statements and mandatory language constructs (enabling class 2 problems). The instructor can also provide a feedback level to change the granularity of steps.

We have also made contributions to the research on using the Ideas framework for programming:

- We have identified which services that were written specifically for Ask-Elle could be reused in a programming tutor for another paradigm. We have identified the issues that should be addressed to generalise these services: providing domain-specific implementations for certain components, such as the predecessor relation and the behaviour of the search mode.
- We have defined both ‘append rules’ and ‘refinement rules’ and use them together to build a programming strategy, whereas Ask-Elle only uses refinement rules. An append rule is a new type of rule which expands a solution instead of rewriting or refining it. In Ask-Elle appending an item to a program such as a function binding is actually refining a hole in a list of holes, in which the holes can already be refined or not.
- We have introduced two new ways for an instructor to influence the feedback: rule ordering based on a preference order in model solutions and the option to generate a solution strategy with different granularity.

7.3 DISCUSSION AND RELATION TO SIMILAR WORK

We use various techniques that have been applied by others, although sometimes using them in a different way. Program transformations on imperative programs are used in several other tutors and assessment tools (Naudé et al., 2010; Rivers & Koedinger, 2013; Xu & Chee, 2003 among others). We use program transformations only for a small number of variations that we do not need to include in our feedback. We also take into account that programs can be incomplete so certain transformations should not be done, such as removing unused variables.

Graphs are used in both programming tutors and assessment tools for multiple purposes. There are graphs that model dependencies between statements representing the control flow of programs (Jin et al., 2012; Naudé et al., 2010; Wang et al., 2007; Xu & Chee, 2003). These graphs are primarily used for matching with student solutions for grading. Graphs with program states and the transitions between them are used by data-driven tutors (Jin et al., 2012; Rivers & Koedinger, 2013). These graphs are built from large quantities of student submissions.

We only use a graph as an intermediate representation to convert to a strategy so we are able to both detect different orders and propose multiple options to students. If we had used dependency graphs to store our entire solution space, we would have had to generate a large number of graphs with little variation because we want to store a large number of variants to include them in the feedback. If we had used graphs to store all our intermediate states, the graphs would have been huge because of the small deltas. Using the strategy language we defer the calculations and are able to provide optimisations. We also have more options that cannot be captured by creating a graph, such as adding a refactoring step at each node.

7.4 FUTURE WORK

New features and improvements. There are many more improvements we could implement in our prototype. In Section 6.3.2 we performed an analysis of a set of actual student programs, in which we identified more variations we would like to recognise by normalisation or by inclusion in the strategy. We also want to expand the annotation capabilities with, for example, a feedback message for multiple statements and using the difficulty of a solution combined with the skill level of a student to personalise the feedback.

Different strategies. Our tutor prescribes a fairly simple way of creating imperative programs: programming line by line with some variation in creating a more complex language construct such as a for loop or a complex boolean expression. We want to investigate if we could use strategies to create programs in a different order, perhaps starting with a basic structure and gradually adding the difficult parts. We have already created an ‘insert rule’ that inserts a given statement at any location in the program but we have not used this rule so far. We would also need instructor annotations to direct this process.

At a low level, we have addressed one of seven elements of the programming process identified by Bennedsen en Caspersen (Bennedsen & Caspersen, 2008): incremental development by taking small steps and testing along the way. We thereby contributed to their appeal to teach students the process towards a solution for a programming problem. We have not yet addressed the issue that programming is generally not a linear process and that refactoring is a valuable process for both novices and experienced programmers. We want to experiment with ‘rewrite rules’ to refactor a student program. For example, a novice programmer could learn how to transform a for loop into a while loop, or vice versa. Another option is to eliminate the use of an unnecessary variable. The model solutions could serve as examples to which we want to refactor.

Improved feedback. When a student asks for a diagnosis after submitting an (almost) finished but incorrect program, we are not able to say much about what went wrong and where it went wrong. It would be an improvement to our tutor if an indication could be given of where the student deviated from the strategy together with precise information on how to repair the program.

Performing transformations, such as the rewriting of an expression into some canonical form, may have consequences for the accuracy of the feedback. We should find a way to adjust the hints to take into account the normalisations that were performed and map back to the student’s solution, as seen in the work of Rivers and Koedinger (Rivers & Koedinger, 2013). Another option is to move some variations to the strategy while keeping the strategy size manageable. We could even investigate if we can dynamically adjust the strategy based on what the student has done so far. We would also have to investigate to what extent the Ideas framework is able to support this.

Language expansion. Currently we support the basic constructs of imperative programming. On our wish list are constructs such as declaring methods, more data types, switch statements, enhanced for loops and do-while statements. Most contemporary imperative languages also support the object-oriented programming paradigm (OOP). OOP supports concepts such as (abstract) classes, objects, interfaces, inheritance and polymorphism. There are not many ITSs that support OOP and we would like to explore how to provide feedback for creating larger object-oriented programs.

Adding the option to declare methods and maybe even objects and classes greatly increases the possibilities for creating exercises. However, we have to add the generation of test cases to provide input for our evaluator to test methods. Furthermore, we have to investigate how incomplete programs can be tested.

Support for multiple imperative languages. So far we have been using the tutor for two imperative languages. Some issues with the shared abstract syntax have already been identified and we would prefer a different solution to embrace the differences between languages. We would have to further investigate how we can easily add new imperative languages to our tutor. We also want to do research into generating parts of the tutor, which was proposed earlier by Jeuring et al. (Jeuring et al., 2012).

Evaluating user experiences. After more features have been included and more attention has been paid to the user interface of the tutor, we want to do experiments with instructors and students. We need to get insight into the findings of students working with the tutor to investigate if the tutor corresponds with their behaviour and if they consider the feedback helpful. We are also interested in how instructors value the tutor and if they are able to deal with creating exercises and annotated model solutions.

8 BIBLIOGRAPHY

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: L. Erlbaum Associates.

Anderson, J., & Skwarecki, E. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM*, *29*(9), 842–849.

Bennedsen, J., & Caspersen, M. (2008). Exposing the Programming Process. In J. Bennedsen, M. Caspersen, & M. Kölling (Eds.), *Reflections on the Teaching of Programming* (Vol. 4821, pp. 6–16). Springer Berlin Heidelberg.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39*(2), 32–36.

Blass, A., Dershowitz, N., & Gurevich, Y. (2009). When are two algorithms the same? *Bulletin of Symbolic Logic*, *15*(2), 145–168.

Bosch, N., Mello, S. D., & Mills, C. (2013). What Emotions Do Novices Experience during Their First Computer Programming Learning Session? In *Artificial Intelligence in Education* (pp. 11–20).

Claessen, K., & Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00 Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (pp. 268–279). ACM Press.

Dadic, T., Stankov, S., & Rosic, M. (2008). Meaningful learning in the tutoring system for programming. In *ITI 2008 - 30th International Conference on Information Technology Interfaces* (pp. 483–488).

Davies, S., Polack-Wahl, J., & Anewalt, K. (2011). A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 625–630). ACM.

Deek, F., & McHugh, J. (1998). A survey and critical analysis of tools for learning programming. *Computer Science Education*, *8*(2), 130–178.

Dehnadi, S., & Bornat, R. (2006). The camel has two humps (working title). Middlesex University, UK.

Dijkstra, E. W. (1975). Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, *18*(8), 453–457.

Gerdes, A. (2012). *Ask-Elle: a Haskell Tutor*. Universiteit Utrecht.

Gerdes, A., Heeren, B., & Jeuring, J. (2010). Properties of Exercise Strategies. *Electronic Notes in Theoretical Computer Science*, *44*, 21–34.

Gerdes, A., Heeren, B., & Jeuring, J. (2012). Teachers and students in charge — Using annotated model solutions in a functional programming tutor. In *7th European Conference of Technology Enhanced Learning, EC-TEL 2012* (pp. 383–388).

Gerdes, A., Jeuring, J., & Heeren, B. (2010). Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM technical symposium on Computer science education - SIGCSE '10* (pp. 441–445). ACM Press.

Gerdes, A., Jeuring, J., & Heeren, B. (2012). An interactive functional programming tutor. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education - ITiCSE '12* (pp. 250–255). ACM Press.

Gomez-Albarran, M. (2005). The Teaching and Learning of Programming: A Survey of Supporting Software Tools. *The Computer Journal*, *48*(2), 130–144.

Guzdial, M. (2004). Programming environments for novices. *Computer Science Education Research*, 127–154.

Hattie, J. (2008). *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. Routledge.

Hattie, J., & Timperley, H. (2007). The Power of Feedback. *Review of Educational Research*, *77*(1), 81–112.

Heeren, B., & Jeuring, J. (2009a). Canonical Forms in Interactive Exercise Assistants. In *MKM 2009: Proceedings of the 8th international conference on Intelligent Computer Mathematics* (Vol. 5625, pp. 325–340).

Heeren, B., & Jeuring, J. (2009b). Recognizing strategies. *Electronic Notes in Theoretical Computer Science*, *237*, 91–106.

Heeren, B., & Jeuring, J. (2011). Interleaving Strategies. *Intelligent Computer Mathematics*, 196–211.

Heeren, B., Jeuring, J., & Gerdes, A. (2010). Specifying Rewrite Strategies for Interactive Exercises. *Mathematics in Computer Science*, *3*(3), 349–370.

Holland, J., Mitrovic, A., & Martin, B. (2009). J-Latte: a constraint-based tutor for Java. In *Proceedings of the 17th International Conference on Computers in Education* (pp. 142–146).

Hong, J. (2004). Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies*, *61*(4), 505–534.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 86–93). ACM Press.

Jeuring, J., Gerdes, A., & Heeren, B. (2012). A programming tutor for Haskell. In *CEFP 2011: Central European Functional Programming School* (Vol. 7241, pp. 1–45).

Jin, W., Barnes, T., & Stamper, J. (2012). Program representation for automatic hint generation for a data-driven novice programming tutor. *Intelligent Tutoring Systems*, 304–309.

Johnson, L. (1986). *Intention-based diagnosis of novice programming errors*. Morgan Kaufmann.

Johnson, W., & Soloway, E. (1985). PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, *11*(3), 267–275.

Jurado, F., Redondo, M., & Ortega, M. (2012). Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice. *Journal of Network and Computer Applications*, *35*(2), 695–712.

Kodaganallur, V., Weitz, R., & Rosenthal, D. (2005). A comparison of model-tracing and constraint-based intelligent tutoring paradigms. *International Journal of Artificial Intelligence in Education*, *15*(2), 117–144.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05* (pp. 14–18). ACM Press.

Le, N., & Menzel, W. (2008). Using constraint-based modelling to describe the solution space of ill-defined problems in logic programming. In *Advances in Web Based Learning–ICWL 2007* (pp. 367–379).

Le, N.-T., Loll, F., & Pinkwart, N. (2013). Operationalizing the Continuum between Well-defined and Ill-defined Problems for Educational Technology. *IEEE Transactions on Learning Technologies*, *6*(1).

Li, J., Pan, W., Zhang, R., Chen, F., Nie, S., & He, X. (2010). Design and implementation of semantic matching based automatic scoring system for C programming language. *Entertainment for Education. Digital Techniques and Systems*, 247–257.

Mason, R., Cooper, G., & Raadt, M. De. (2012). Trends in introductory programming courses in Australian universities: languages, environments and pedagogy. In *Proceedings of the Fourteenth Australasian Computing Education Conference* (Vol. 123, pp. 33–42). Australian Computer Society, Inc.

McCracken, M., Almstrum, V., & Diaz, D. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125–180.

Miller, P., Pane, J., Meter, G., & Vorthmann, S. (1994). Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments*, *4*(2), 140–158.

Mitchell, N., & Runciman, C. (2007). Uniform Boilerplate and List Processing. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell* (pp. 49–60). ACM.

Naudé, K. a., Greyling, J. H., & Vogts, D. (2010). Marking student programs using graph similarity. *Computers & Education*, *54*(2), 545–561.

Nunez, A., Fernandez, J., & Carretero, J. (2010). M-PLAT: Multi-Programming Language Adaptive Tutor. In M. B. Rosson (Ed.), *Advances in Learning Processes* (pp. 125–138). InTech.

Nwana, H. (1990). Intelligent tutoring systems: an overview. *Artificial Intelligence Review*, 251–277.

Paterson, R. (2003). Arrows and computation. *The Fun of Programming*, 201–222.

Pausch, R., & Kelleher, C. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, *37*(2), 83–137.

Pears, A., Seidman, S., Malmi, L., & Mannila, L. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, *39*(4), 204–223.

Pieterse, V. (2013). Automated Assessment of Programming Assignments. In *CSERC '13 Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research* (pp. 45–56).

Pillay, N. (2003). Developing intelligent programming tutors for novice programmers. *ACM SIGCSE Bulletin*, *35*(2), 78–82.

Rivers, K., & Koedinger, K. (2013). Automatic Generation of Programming Feedback: A Data-Driven Approach. In *The First Workshop on AI-supported Education for Computer Science* (pp. 50–59).

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172.

Singh, R., Gulwani, S., & Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *34th annual ACM SIGPLAN conference on Programming Language Design and Implementation* (pp. 15–26).

Soloway, E., & Spohrer, J. C. (1989). *Studying the novice programmer*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Song, J., Hahn, S., Tak, K., & Kim, J. (1997). An intelligent tutoring system for introductory C language course. *Computers & Education*, *28*(2), 93–102.

Spiro, R., Coulson, R., Feltovich, P., & Anderson, D. (1988). Cognitive flexibility theory: Advanced knowledge acquisition in ill-structured domains. In *Tenth Annual Conference of the Cognitive Science Society Proceedings* (pp. 640–653). Hillsdale, NJ: L. Erlbaum Associates.

Sykes, E. (2005). Qualitative Evaluation of the Java Intelligent Tutoring System. *Journal of Systemics, Cybernetics and Informatics*, *3*(5), 49–60.

Sykes, E., & Franek, F. (2003). A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java (TM). In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education* (pp. 78–83).

Vanlehn, K. (2006). The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education*, *16*(3), 227–265.

Vanneste, P. (1994). *A reverse engineering approach to novice program analysis*. KU Leuven Campus Kortrijk.

Vujošević-Janičić, M., Nikolić, M., Tošić, D., & Kuncak, V. (2013). Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology*, *55*(6), 1004–1016.

Wadler, P. (1998). A Prettier Printer. *Journal of Functional Programming*, 223–244.

Wang, T., Su, X., Wang, Y., & Ma, P. (2007). Semantic similarity-based grading of student programs. *Information and Software Technology*, *49*(2), 99–107.

Weragama, D., & Reye, J. (2013). The PHP Intelligent Tutoring System. In *Artificial Intelligence in Education* (pp. 583–586). Springer Berlin Heidelberg.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, *14*(4), 221–227.

Xu, S., & Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, *29*(4), 360–384.