Robin McFarland

Giving Programming Exercises Adaptive Difficulty

Computer Science Tripos – Part II

Homerton College

2017

Proforma

Name:	**Robin McFarland**
College:	**Homerton College**
Project Title:	**Giving Programming Exercises Adaptive Difficulty**
Examination:	**Computer Science Tripos – Part II, July 2017**
Word Count:	**TBC**
Project Originator:	Mr Michael B. Gale
Supervisor:	Mr Michael B. Gale

Original Aims of the Project

Work Completed

Special Difficulties

Declaration

I, Robin McFarland of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Acknowledgements

Chapter 1

Introduction

Chapter 2

Preparation

- First attempt using Strings
- Why I had to make my own language
- Research into other projects etc, like Arjen and the PhD

Chapter 3

Implementation

("it feels a lot like you are explaining the code with the expectation that the reader is looking at the code and reads your description alongside it, when you should really be explaining how it works and what the intuition is (note: explaining how something works is not the same as explaining the code)")

("You should be approaching the write-up from the perspective of "how do I explain the problems and solutions to someone who has never seen anything about this project before", don't go through the code, pick parts, and describe those.")

In this chapter I detail the work completed. I begin by describing how I implemented MiniJava, (a small imperative programming language based on Java), as well as a parser for MiniJava's grammar. I then describe the algorithm that allows the addition and removal of "blanks" within a MiniJava program. I then describe how a programming exercise is conceptualised in the abstract sense, and give an example of an implementation of a specific question. Finally, I describe the two peripherals I developed to interact with the language; the `ExerciseSetter` and the Graphical User Interface (GUI).

3.1 Designing the Abstract Syntax Tree for MiniJava

In this section I describe how I designed the MiniJava language.

Since part of my aim was to demonstrate a possible method for teaching Java, I wanted the language I was using to be as close to Java as possible. However, re-implementing all of Java would have been too much unnecessary complexity, so I designed MiniJava with reference to only a small subset of the Java 8 grammar [2, p.714]. Because my eventual aim was to replace parts of MiniJava programs with blanks for a student to fill in, storing MiniJava programs as Abstract Syntax Trees (ASTs) was the obvious choice. Since nodes in the AST represent constructs in the source code, replacing a node in the AST with a notional "blank" node would be representative of making parts of the source code blank.

Each class in my implementation represents a different production rule in the abstract syntax for MiniJava, found in Appendix A in the form of a parser specification. My first task was to differentiate between MiniJava expressions and MiniJava statements. This distinction is important as they must be used in different places: for example, in the production rule `WHILE parExpression statement` representing a `while` loop, `parExpression`

may be substituted with any MiniJava expression (surrounded by parentheses), while `statement` may be substituted with any MiniJava statement. The reverse is not allowed however, meaning that `statement` may not be replaced by a MiniJava expression and vice versa. To encode this distinction, I made the interfaces `Expression` and `BlockStatement` (which all MiniJava expressions and statements implement respectively), and the abstract classes `ExpressionBase` and `StatementBase` (which all MiniJava expressions and statements extend respectively). For justification of why both the interfaces and the abstract classes are required, see section 3.5.

As important as the distinction between expressions and statements is, there do exist some expressions that can be used in statements. For an example, the MiniJava code `i = 5;` is a statement because it ends in a semicolon, ";", but this statement consists entirely of the expression `i = 5`. This sort of expression within a statement construct is not possible with all expressions, for example the code `i + 4;` would not be valid MiniJava. To make the distinction between expressions which may be used in statements like this and those which may not, I made the interface `StatementExpression`. This interface is only implemented by those expressions which can be used in a statement this way.

A feature of MiniJava is the concept of operator precedence, which is used to disambiguate an otherwise ambiguous grammar of expressions. Without the concept of operator precedence, the code `1 + 2 * 3` is ambiguous, as it can be read as either `(1 + 2) * 3` which equates to 9, or as `1 + (2 * 3)` which equates to 7. Since a single arithmetic expression cannot have more than one value, a decision must be made as to which of these interpretations is declared valid. The decision taken aligns with the standard arithmetic order of operations, such that the answer of 7 is the correct answer in this case. This example demonstrates that the multiplication operator, "`*`", has a higher precedence than the addition operator, "`+`". All the possible operators in MiniJava are found within this precedence hierarchy, and I needed some way to encode this. The method I chose for this is the natural choice, since it is suggested by the Java grammar itself [2, p.723]. The production rule for `AdditiveExpression` is as follows:

```
AdditiveExpression:
    MultiplicativeExpression                                (1)
    AdditiveExpression + MultiplicativeExpression           (2)
    AdditiveExpression - MultiplicativeExpression           (3)
```

and the production rule for `MultiplicativeExpression` is as follows:

```
MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    // Note that the modulus operator, %, was removed from MiniJava
```

This means that for any expression A `+` B, both A and B must only contain subexpressions that use operators with precedence at least as high as `+` and `-`, and that for any expression A `*` B, neither A nor B can contain subexpressions that use operators with precedence lower than `*` and `/`. This in turn indicates that an expression such as `1 + 2 * 3 + 4` can only be interpreted as `1 + (2 * 3) + 4`, and an expression such as `1 * 2 + 3 * 4` can only be interpreted as `(1 * 2) + (3 * 4)`. I wanted an operator precedence for MiniJava that is hardcoded within the grammar of MiniJava itself, just as the operator precedence of Java is hardcoded within the grammar of Java. The production rules for Java shown above suggest a natural implementation of the class that represents addition expressions:

```
class AddExpr {
    private boolean isPlus;
    private AddExpr leftSide;
    private MultExpr rightSide;

    ...
}
```

Here, the `isPlus` field records whether the expression is an addition or a subtraction, `leftSide` and `rightSide` store the left and right operands of the expression, and `MultExpr` is the class representing multiplication expressions. What has been implemented so far encapsulates parts (2) and (3) of the `AdditiveExpression` production rule above, but not part (1). This part is encapsulated by making the class `MultExpr` extend the class `AddExpr`, such that an instance of `MultExpr` may always be used in place of an instance of `AddExpr`. Note that since multiplication operators are higher in the operator precedence hierarchy, their representations are lower in the object hierarchy. This then is the solution to the operator precedence problem: to enforce an operator precedence hierarchy, first enforce an inverse object hierarchy in their representative classes, and then choose the type of each expression's operands to match the relevant terminal symbols in the relevant production rule.[1]

[1]Note that if you were to look at my implementation of `AddExpr`, `leftSide` and `rightSide` would both be `int`s, not `AddExpr` and `MultExpr` instances. The reason behind this is explained in section 3.5.

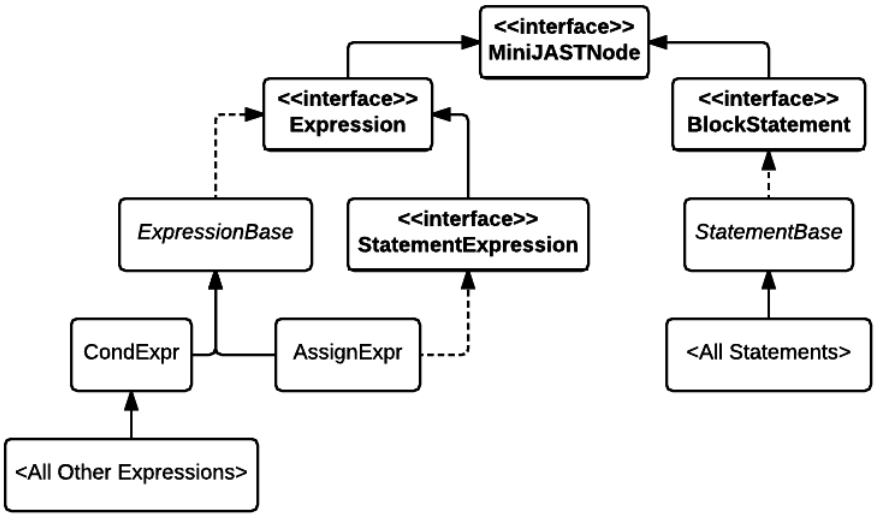

Figure 3.1: The UML diagram showing the top of the object hierarchy for `MiniJASTNode`s. I haven't yet used this in my explanation, but it will probably go at the end of my last paragraph, just to collect together everything I've said so far.

Still to explain:

1. The PrimType and Type classes (paragraph on how the types of expressions are propagated)
2. SingleWordStmnt (paragraph on why four statements are stored in one enumeration)
3. AssignOp and RelationOp (paragraph comparing the `isPlus` field used when the operator is either + or - and the AssignOp enum for when the possible operator is =, +=, -=, *= or /=)
4. AssignLHS (paragraph on the extra classification of expressions as being able to be assigned to)
5. ArrayAssignRightSide and VarDeclarator (paragraph on the evolution of the class VarDeclarator, and a paragraph on the extra classification of expressions as being able to be assigned to an array)

3.2 Interpreting the language

In this section I explain how I wrote an interpreter for MiniJava.

Remove all references to "language features", they are "productions in the abstract syntax".

(Distinguish between the tree and the interpreter.)The `evaluate` method takes a `Context` object and returns a `ReturnValues` object (Use bottom explanation, i.e. "The evaluate method implements an interpreter for MiniJava. Before looking at its implementation, let us introduce some helper classes that are needed by the implementation of evaluate."). (Give more intuition about what the classes do!)A `Context` object stores three maps: (Give the names and then the descriptions) one map of names to `Types`, one of names to values, and one of names to depths. A `Type` object stores the primitive type (`boolean`, `char`, `int`, or `double`) associated with the type, and whether the type is an array type or not, and also the size of the type (this is not 1 only if the value is an array). The `Context` object allows the use of variables. If an identifier has been declared in this scope then it will have an entry in the `namesToTypes` and `namesToDepths` maps. If it has also been defined, then it will also have an entry in the `namesToValues` map. Thus, on variable initialisation, we can check if the identifier has already been used in this scope, and also update its value on assignment. The `namesToDepths` map is used to keep track of scope, as every time we leave a block (and thus our depth decreases), we remove all the entries from these maps where the depth value is the depth just left.

(Explain intuition) `ReturnValues` is an abstract class, with one implementation for each of the four possible primitive types, and another for an array value. Each of the four implementations stores a value of the appropriate type. `ReturnValues` objects are how information is passed up the program tree during evaluation. When a node needs to be evaluated, it will evaluate its sub-nodes and use the `ReturnValues` objects passed up to extract the necessary information.

(Bottom up explanation) Since array accesses are a special case, they require another four implementations of `ReturnValues` that stores not only the value but also the name of the array it was accessed from and the index it was found at. The reason array accesses require special treatment is that they can be used on the left hand side of assignments. Consider the following code and imagine that array accesses did not get special treatment:

```
a[i++] = i;
```

(Explain evaluation better in general) In the evaluation of this code, `a[i++]` is first evaluated as an array access to ensure that `a` and `i` are both initialised, and also get the value of `a[i++]`. Thus the `ReturnValues` object returned is a `ReturnValuesInt` that stores the value of `a[i++]`. The `i` on the right hand side is then evaluated, and the assignment is ready to take place. (Fluffy, don't use "recognises")The system recognises that there is an assignment to an array, and attempts to update the value in the `namesToValues` map, but the system has no way to know what index to store the value at. The `ReturnValues` object doesn't store it, the `ArrayAccess` object itself stores it as an `Expression` which must be evaluated to get a value, which would not only be wrong, but also change the value of `i` again. Thus there is no way to update an array without having a special version of `ReturnValues` used for array accesses that also store the index it was meant to be stored at.

Looking now at the `BlockStatement` interface(Show it again), we see the three methods `execute`, `executeStart` and `stringRepr`. In some ways these are analogous to the methods found in the `Expression` interface. `stringRepr` returns a `String` representation of the statement, but also requires a `blocksDeep` parameter to know how many tab characters to put in front of the text. The `execute` method takes a `Context` object and a `depth` parameter and returns a `FlowControl` value. The `depth` parameter is used in scope management by the `removeDecsAtDepth` method in `StatementBase`. `FlowControl` is an enumeration determining what action must be taken after a statement is executed, and can take one of four values: `BREAK`, `CONTINUE`, `RETURN` and `NONE`. `executeStart` is simply a helper method for starting an execution with the appropriate initial depth value.

Whenever something goes wrong during execution, for example there is a type error or a variable is used before it is initialised, a custom exception is thrown. Information about the mistake that was made can be extracted from the exception type and from the message passed with it.

3.3 The Parser

In this section I explain how a parser for MiniJava was implemented using the ANTLR parser generator [1], and how a MiniJava program tree can be built using it.

("I based the parser grammar on an existing one for Java". Also justify this) The GitHub user antlr that owns the antlr4 project containing ANTLR has another project called grammars-v4[2] that contains lots of example grammars that can be used with ANTLR, including one for Java. I was able to modify this grammar file, in a similar way to how I modified the Java grammar when designing MiniJava, to make a parser generator for MiniJava, (I didn't make a parser generator! A parser generator made my parser to which I added actions) which can be found in Appendix A. When you (Not you) use antlr4 on the grammar file, the tool generates a number of files including a `<GrammarName>BaseVisitor`. By extending this class, one can add actions to the parser. By overriding the appropriate methods in this class, I was able to implement a parser that, given valid MiniJava code, could build a MiniJava program tree. (Talk more about how the parser works)

3.4 The Abstract Question

(Make title more descriptive, and fix the capitals)

In this section I explain the formulation of the abstract question and give an example of how a concrete question can be derived from it. (Explain this)

("The Implementation chapter should not just be a description of every class in your implementation, but an explanation of the key problems you had to solve and how you solved them. From reading your Implementation so far, I have no good idea for why any of these components exist, how they fit together, and what problems they solve.")

[2]https://github.com/antlr/grammars-v4

The abstract class `AbstractPExercise` represents an abstract programming exercise. It has fields including `question` (a `String` containing the question shown to students), `solution` (a `BlockStatement` object containing the model solution for the problem) and `baseDifficulty` (a measurement of the relative difficulty of this exercise compared to others, imposes a natural ordering on exercises in terms of their difficulty), and abstract methods including `setUp` (where the model solution is built) and `checkSolved` (where the current response is run and the system can determine whether the student has solved the problem). The other fields are required for either actually running the answer, or for adding blanks to and removing blanks from the model solution.

An example implementation of these abstract methods is found in `FactorialExercise`, (The code won't be provided so if referenced quote it) an exercise in which the student is asked to use a while loop to calculate the factorial of some integer `n`, which is supplied on construction of the object. The overridden `setUp` method constructs the model solution (Explain), which represents the code:

```
int total = 1, n = N;
while (n > 1) {
    total *= n--;
}
```

where `N` is the number supplied at construction. This code calculates `N!` and stores it in `total` as required. The `solution` field from `AbstractPExercise` is filled with the `Block` object that contains the code. Note also that `FactorialExercise` defines a new `totalId` field of type `Id`, which represents the variable `total`. This is used in the overriden `checkSolved` method, where the value of `total` is checked against the correct answer of `N!`.

This demonstrates one of the two ways that solutions can be checked: either the value of a variable can be checked by storing its `Id` in a field and accessing its value later, or an output stream could be printed to using the standard `System.out.println` method that has been included in MiniJava for convenience and later checked.

In this section I have presented the class `AbstractPExercise` and given an example implementation of it.

3.5 Adding and Removing Blanks

This section explains the implementation of `FillableBlank` and its two implementations, before describing the algorithm that adds a blank to a MiniJava program that can later be filled in by a student, and the algorithm that replaces a blank with the original code snippet.

`FillableBlank` is an abstract class that encapsulates the behaviour of having a unique id and storing the number of nodes that this blank replaces. It is inherited by both `FillableBlankExpr` and `FillableBlankStmnt`. `FillableBlankExpr` implements `Expression` and `FillableBlankStmnt` implements `BlockStatement`, such that both are also `MiniJASTNodes`. Each stores a `MiniJASTNode` of the appropriate classification that is either `null`, or represents student code.

The algorithm that adds a blank to a MiniJava program is found in the `addBlank` method of the `AbstractPExercise` class. The precondition for the algorithm is that the `solution` field contains a MiniJava program with a certain percentage blank. The postcondition is that either the solution was entirely blank, in which case the algorithm returns false, or the `solution` field contains the same MiniJava program but with a larger percentage blank, the head of the `replacedNodes` stack is the replaced node, and the head of the `replacedNodeTreeIndices` stack is a stack containing the indices describing the path to the blank that has just been added. The increase in percentage blank will be the smallest increment possible at that time.

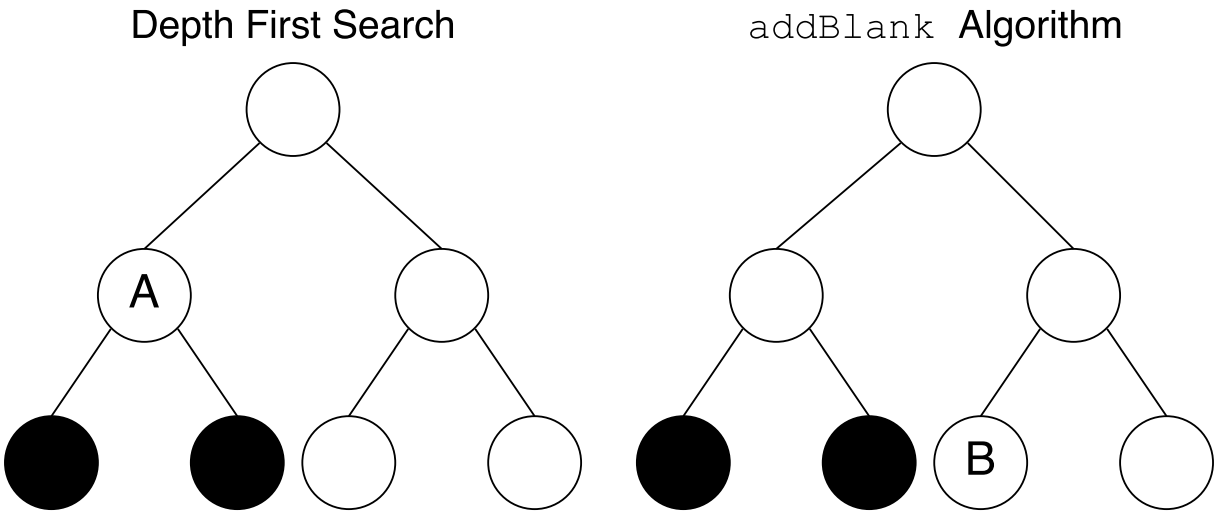

Figure 3.2: A figure demonstrating the different behaviour of a depth first search algorithm and my `addBlank` algorithm.

(Reverse Breadth First Search maybe?)

In order to ensure the increase in percentage blank is the smallest increment possible, the algorithm always selects a leaf to replace with a blank, where here a leaf is defined as a node that either has no children, or all its children are already blank. This desire to reach leaves quickly motivates the use of a depth first search algorithm, but using it without modification presents a problem. Figure 3.2 shows the difference between the selection made by a depth first search algorithm and the selection that should be made. In the diagram, the black nodes represent blanks, and the white nodes represent nodes in the tree that we might choose to make blank. We see that a normal depth first search algorithm would select node A next, whereas `addBlank` should select node B. As such, we introduce the notion of "marking" a node. When we detect that all of node A's children are blank, we declare that that node is a leaf and we mark it. At this point, the algorithm is looking to replace only unmarked leaves, so on this pass node B will be selected, and on the next pass node A will once again not be selected. Only when all the leaves at this depth have been made blank will node A be made blank.

The nodes are stored in a stack as they are dealt with. However, because more information is required as the algorithm progresses, the parent of the current node being considered, the index of the current node within its parent's `subnodes` field, and whether the current parent has all its children blank are also stored in stacks. We start by adding the root node of the solution tree to the nodes stack. When we reach a node, it falls in to one of four categories:

This node's children have already been searched: In this case, we need to check whether all of this node's children are blank. If they are, then technically this node is a leaf, but we don't want to replace it next walk of the tree so we mark it accordingly. Either way, we move on to the next node.

This node is a leaf: If this node is marked appropriately, then we need to go about replacing it with a blank. First we will check if this node is in fact the root node, in which case we can replace the entire solution with a blank. Either way, we store the replaced node itself and the path through the tree to its location in terms of indices so that it can later be reinserted if necessary. If the node to be replaced is an `Expression`, we replace it with a blank expression, otherwise we replace it with a blank statement. Either way we return true, as a replacement has been made.

If instead the node is not marked appropriately, then we move on to the next node after recording that the current parent has a child that is not blank.

This node is blank: If this node is the root of the tree, then the entire solution is blank, and we can't add any more blanks, so we return false. Otherwise, we simply move on to the next node.

This node is none of the above, and thus has children that need searching: In this case we need to register that we are increasing our depth of search. This means we add the current node to the parents stack, we add a new 0 to the indices stack and we add a true to the `childrenBlank` stack after first recording that the old parent has a child that isn't blank. We then add all this node's children to the nodes stack in reverse order, so that they emerge from the stack in the correct order.

Within the `while(true)` loop the entire solution tree is walked. Remove reference to code that can't be seen!)If a result has not been reached in one iteration of the loop, then all the leaves are marked for not being selected this walk of the tree, so we change the marking we are looking for and walk the solution tree again.

The algorithm that removes a blank from a MiniJava program is found in the `removeBlank` method of the `AbstractPExercise` class. It is somewhat simpler than the `addBlank` method discussed before, as it simply makes use of all the information that has to be recorded during the execution of `addBlank`. If the `replacedNodes` stack is empty then there are no blanks to be removed, so we return false. If instead the head of the `replacedNodeTreeIndices` stack is empty, then we are replacing the whole solution with the head of the `replacedNodes stack`. Otherwise, we need to follow the path described in the head of the `replacedNodeTreeIndices` stack (once it has been reversed, since it was stored reversed during `addBlank`), to find the location of the blank that needs to be replaced by the head of the `replacedNodes`. We need to make sure that the parent of this node is no longer considered a leaf, and we also need to make sure that if the marking we are searching for was just changed by the last `addBlank` invocation (i.e. the index we have is always the largest possible) then we change it back now.

This section first explained how the notion of a fillable blank was added to MiniJava, before describing the implementations of the algorithms that introduce them to and remove them from a MiniJava program.

3.6 The `ExerciseSetter`

In this section I describe how I made use of the language and its features to design the `ExerciseSetter`, one possible way in which students might interface with exercises designed to facilitate the learning of MiniJava.

The `ExerciseSetter` stores a list of possible exercises in order of increasing difficulty. The initial exercise to be delivered can be easily adjusted by changing the field `INITIAL_EX`, and the current exercise index is also stored, along with a reference to that exercise. To measure the difficulty of the exercise and the performance of the students, the number of attempts at a solution made, the number of nodes in the solution, and the number of blanks added are also stored. A lot of the code in this class goes toward keeping these values consistent. The `ExerciseSetter` also stores a reference to an `OutputStream`, where all the output can be written to. This allows the `ExerciseSetter` to be plugged in to other peripherals, like the GUI described below. Finally, it also stores a reference to the parser.

A lot of the methods in this class are helper methods, simply passing messages between the exercise and the student. The important ones are `fillBlank`, `reportPerformance`, and `adjustQuestion`.

`fillBlank` has two method signatures, one that takes a `MiniJASTNode`, and one that takes a `String`. The one that takes a `MiniJASTNode` is trivial and thus unimportant, but the one that takes a `String` is more interesting. This method makes use of the parser in an interesting way, as there is of course some ambiguity in the language. The specific ambiguity comes from certain code snippets having the potential to be both `Expression`s and `BlockStatement`s. An example of this is the code snippet `total = 1`. This can be parsed as an assignment, and would thus be classed an `Expression`, or a variable declaration, where the surrounding context might be `int total = 1;`, which would categorise this snippet as a `BlockStatement`. To make the decision, the `ExerciseSetter` first checks if the blank being filled represents an `Expression` or a `BlockStatement`, and sets the entry point for the parser appropriately.

`reportPerformance` is a possible implementation of a performance heuristic. Performance is based on the number of attempts the exercise took to solve, and the number of nodes in the solution compared with the model solution. The more attempts required and the more nodes in the solution, the worse the performance. A negative result here indicates that the next exercise should be easier, while a positive result indicates it should be harder.

`adjustQuestion` is a possible implementation of how the performance might influence the difficulty of the next problem. If the next exercise needs to be harder, then the `ExerciseSetter` adds blanks to the solution until either the exercise is hard enough, or the entire solution is blank. If the latter occurs, then the `ExerciseSetter` attempts to present a harder problem, and calculate how much of it should be blank. If instead the exercise needs to be easier, then the `ExerciseSetter` removes blanks from the solution until either the exercise is easy enough, or there are no blanks in the solution. If the latter occurs, then the `ExerciseSetter` attempts to present an easier problem and calculates how much of it should be blank.

In this section I have detailed one of the ways that students might interface with MiniJava through the `ExerciseSetter`. This is only an example of a possible use for the tools, but shows how powerful they can be.

3.7 The GUI

In this section I describe the implementation and function of the GUI peripheral.

The GUI demonstrates some of the possible uses of this set of tools. It shows how a teacher might manually set the difficulty of a particular exercise and how a student could fill in the blanks and run their solution. An example screen from the GUI is shown in Fig 3. The top slider determines the problem that is presented: the further to the left the slider is, the easier the problem will be. The bottom slider affects the difficulty of that problem: the further it is to the left, the fewer blanks there will be and thus the easier the problem will be to solve. When the “Generate” button is pressed, the corresponding exercise will be displayed in the box on the right.

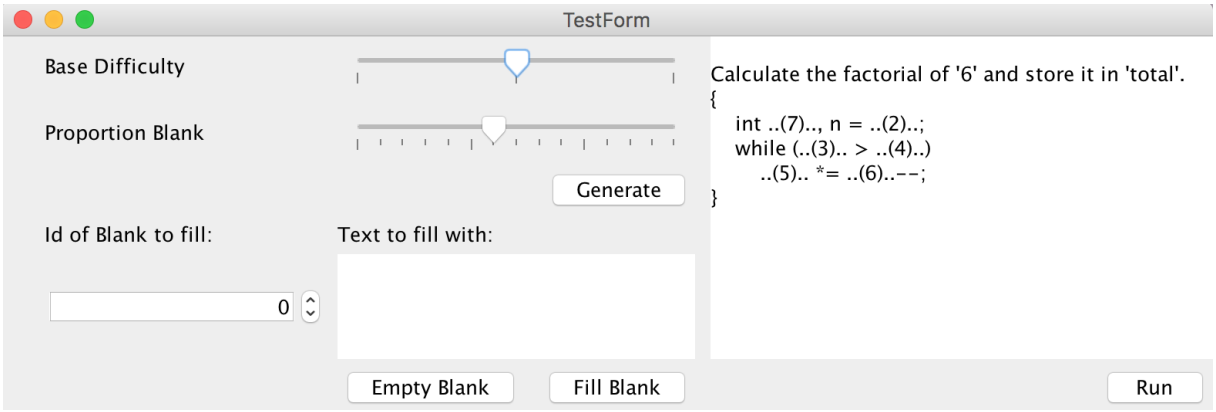

Figure 3.3: An example screenshot taken of the GUI peripheral.

When an exercise is generated, the blanks within the solution that need to be filled are shown as numbers in brackets surrounded by ellipses. To fill one of the blanks, the user must enter its number in the spinbox, and then enter the code with which to fill the blank in the text box, before pressing the button marked "Fill Blank". A blank with a given number can be emptied by entering its number into the spinbox and pressing the button marked "Empty Blank". The solution, i.e. the text in the large box on the right, can be executed at any time by pressing the "Run" button. The GUI will report whether the solution is correct or not.

The GUI was designed using the IntelliJ UI Designer plugin[3]. This plugin allowed me to drag and drop components into place on the form before programming their functionality with the adjoining bound class. The class stores a reference to an `ExerciseSetter` object, which is where all the data comes from, and where the commands from the user are delivered to. It also makes use of the `ExerciseSetter`'s `setOutput` method by giving it a `ByteArrayOutputStream` that can then be read as text to be displayed. There is some error handling involved in that when something goes wrong, e.g. the user attempts to fill a blank with a number that isn't in the solution, or a solution run raises an error, the error is propagated up from the `ExerciseSetter` and displayed in the GUI.

This section described the function and then the implementation of the GUI peripheral.

In this chapter I have described what I have implemented and how I have done it. I started by giving the design of the language MiniJava, explaining some of these choices by introducing the parser and the concept of "blanks" within a MiniJava program. From there I explained how a programming exercise is conceptualised in the abstract sense, and gave an example of an implementation of a specific question. I finished by giving two possible uses for the set of tools, the `ExerciseSetter` and the Graphical User Interface (GUI).

Estimated Word Count: 3765

[3]https://www.jetbrains.com/help/idea/2017.1/swing-designing-gui.html

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

[1] Author. *The Definitive Antlr 4 Reference*. Publisher, Year.

[2] Author. *Java 8 Spec*. Publisher, Year.

Appendix A

The Parser

```
grammar MiniJava;

// STATEMENTS / BLOCKS

// entry point
entry
    : block [true]          # blockEntry
    | blockStatement+       # blockStatementsEntry
    | statementTop          # statementEntry
    | expression            # expressionEntry
    ;

block [boolean isOuter]
    :   LBRACE blockStatement* RBRACE
    ;

blockStatement
    :   primitiveType variableDeclarators SEMI  # localVariableDeclaration
    |   statement                               # makeStmnt
    |   variableDeclarator                      # makeVarDec
    ;

statementTop
    :   statement                       # stmnt
    |   statementNSI                    # stmntNSI
    ;

statement
    :   block [false]                                       # makeBlock
    |   IF parExpression statement                          # makeIf
    |   IF parExpression statementNSI ELSE statement        # makeITE
    |   FOR LPAREN forInit? SEMI expression?
                SEMI expressionList? RPAREN  statement      # makeFor
    |   WHILE parExpression statement                       # makeWhile
    |   statementNTS                                        # makeStatementNTS
    ;

statementNSI
```

```
    :  block [false]                                          # makeBlockNSI
    | IF parExpression statementNSI ELSE statementNSI         # makeITENSI
    | FOR LPAREN forInit? SEMI expression?
            SEMI expressionList? RPAREN statementNSI          # makeForNSI
    |   WHILE parExpression statementNSI                      # makeWhileNSI
    |   statementNTS                                          # makeStatementNTSNSI
    ;

statementNTS
    :   DO statement WHILE parExpression SEMI   # makeDo
    |   RETURN SEMI                             # return
    |   BREAK SEMI                              # break
    |   CONTINUE SEMI                           # continue
    |   SEMI                                    # empty
    |   expressionStatement SEMI                # makeStmntExpr
    ;


forInit
    :   primitiveType variableDeclarators          # forInitLVD
    |   expressionList                             # forInitExprs
    ;

// EXPRESSIONS

parExpression
    :   LPAREN expression RPAREN
    ;

expressionList
    :   expression (COMMA expression)*
    ;

expressionStatement
    :   expression
    ;

expression // Most binding comes first!
    :   Identifier                                              # makeID
    |   expression LBRACK expression RBRACK                     # arrayAccess
    |   parExpression                                           # makeBracketed
    |   literal                                                 # makeLiteral
    |   expression (op=INC | op=DEC)                            # postInc
    |   (op=ADD|op=SUB|op=INC|op=DEC) expression               # preIncEtc
    |   BANG expression                                         # makeNot
    |   expression (op=MUL|op=DIV) expression                   # multExpr
    |   expression (op=ADD|op=SUB) expression                   # addExpr
    |   expression (op=LE | op=GE | op=GT | op=LT) expression   # relationalExpr
    |   expression (op=EQUAL | op=NOTEQUAL) expression          # eqExpr
    |   expression AND expression                               # andExpr
    |   expression OR expression                                # orExpr
    |   <assoc=right> expression QUESTION
            expression COLON expression                         # condExpr
```

```
    |   <assoc=right> expression
        (   op=ASSIGN
        |   op=ADD_ASSIGN
        |   op=SUB_ASSIGN
        |   op=MUL_ASSIGN
        |   op=DIV_ASSIGN
        )
        expression                                              # assignExpr
    ;

// VARIABLES AND LITERALS

variableDeclarators
    :   variableDeclarator (COMMA variableDeclarator)*
    ;

variableDeclarator
    :   Identifier LBRACK RBRACK (ASSIGN variableInitializer)?  # arrayVarDec
    |   Identifier (ASSIGN variableInitializer)?                # singleVarDec
    ;

variableInitializer
    :   arrayInitializerValues                                  # arrayInitVals
    |   arrayInitializerSize                                    # arrayInitSize
    |   expression                                              # initExpr
    ;

arrayInitializerValues
    :   LBRACE variableInitializer (COMMA variableInitializer)* (COMMA)? RBRACE
    ;

arrayInitializerSize
    :   NEW primitiveType LBRACK expression RBRACK
    ;

primitiveType
    :   BOOLEAN
    |   CHAR
    |   INT
    |   DOUBLE
    ;

literal
    :   IntegerLiteral
    |   FloatingPointLiteral
    |   CharacterLiteral
    |   BooleanLiteral
    ;

// LEXER

//  3.9 Keywords
```

```
BOOLEAN       : 'boolean';
BREAK         : 'break';
CHAR          : 'char';
CONTINUE      : 'continue';
DO            : 'do';
DOUBLE        : 'double';
ELSE          : 'else';
FOR           : 'for';
IF            : 'if';
INT           : 'int';
NEW           : 'new';
RETURN        : 'return';
WHILE         : 'while';

// 3.10.1 Integer Literals
// 3.10.2 Floating-Point Literals
// 3.10.3 Boolean Literals
// 3.10.4 Character Literals
// 3.11 Separators
// Sections removed for clarity

LPAREN          : '(';
RPAREN          : ')';
LBRACE          : '{';
RBRACE          : '}';
LBRACK          : '[';
RBRACK          : ']';
SEMI            : ';';
COMMA           : ',';
DOT             : '.';

// 3.12 Operators

ASSIGN          : '=';
GT              : '>';
LT              : '<';
BANG            : '!';
QUESTION        : '?';
COLON           : ':';
EQUAL           : '==';
LE              : '<=';
GE              : '>=';
NOTEQUAL        : '!=';
AND             : '&&';
OR              : '||';
INC             : '++';
DEC             : '--';
ADD             : '+';
SUB             : '-';
MUL             : '*';
DIV             : '/';

ADD_ASSIGN      : '+=';
```

```
SUB_ASSIGN      : '-=';
MUL_ASSIGN      : '*=';
DIV_ASSIGN      : '/=';

//  3.8 Identifiers (must appear after all keywords in the grammar)
//
// Whitespace and comments
//
// Sections removed for clarity
```

Appendix B

Project Proposal

Computer Science Tripos – Part II – Project Proposal

A System for Giving Programming Exercises Adaptive Difficulty

R. J. McFarland, Homerton College
Originator: M. B. Gale
20 October 2016

Project Supervisor: M. B. Gale
Director of Studies: Dr J. Fawcett
Project Overseers: Dr D. J. Greaves & Prof J. G. Daugman

Introduction

Different people learn at different speeds, and learn some things more quickly than others. When a group of students are given a set of programming exercises, this typically isn't taken account of. The aim of this project is to design a system that varies the difficulty of a programming exercise depending on how the individual has performed completing previous exercises.

The difficulty of an exercise can be measured using two metrics: how complex the problem being solved is and how much code the student is expected to write. The system will vary both of these to adapt the difficulty of proposed exercises automatically (i.e. the system will not look for preprogrammed mistakes, but rather interpret the mistakes made and react accordingly). When an exercise is completed to a sufficient standard, a similar exercise will be given, but expecting more code to be written. When the system has determined that the problem has been mastered, a more difficult problem will be presented, requiring less code to be written again.

Resources required

I shall use my own Macintosh laptop for the majority of this project. Backup will be to GitHub and to a 5TB hard drive I keep in my room. I shall make use of the Java standard library, as well as the JavaFX library for the graphical user interface element. I require no special resources.

Starting point

I am able to program in Java, and have used JavaFX before for the Part IB group project. During my A-Levels I wrote a program to test the maths abilities of Year 4 students, which involved the programmatic generation of various kinds of maths questions.

Work to be done

The work for this project can be split up into the following sub-projects:

1. A representation of a programming exercise must be coded up to allow the final product to manipulate them.
2. A heuristic that measures the difficulty of a given exercise must be chosen and implemented.
3. A heuristic that assesses how well a "student" has completed an exercise must be chosen and implemented.
4. I must devise a system, using the previous three items, that determines which exercise should be given to the student next, as well as how much of the required code is already filled in, based on the student's performance completing previous exercises.
5. A Graphical User Interface should be designed to enable a user to manually adjust the content of a given programming exercise.
6. In order to test the program, a student will be simulated by creating a system that tells the program what errors were made where and how long the exercise took to be solved, so that the program's response to various stimuli can be measured.

Success Criteria

The project will be considered completed when: