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

MiniJava is a language I have developed.

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

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 [1, p.714]. 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.

"Are there any limitations of using antlr and how it compared to other parser generators / techniques"

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 MiniJava's Abstract Syntax Tree

In this section I describe the implementation of the MiniJava language.

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. It is important to differentiate between MiniJava expressions and MiniJava statement 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. This distinction is made by 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). Both interfaces and abstract class are used here because of the need to include blanks in the grammar. There are two classes representing blanks:

`FillableBlankExpr` and `FillableBlankStmnt`. The first of these should be a MiniJava expression and the second should be a MiniJava statement, meaning they should extend `ExpressionBase` and `StatementBase` respectively, but both of them must also extend the abstract class `FillableBlank` (this is explained on page 17). Since classes in Java cannot extend more than one base class, the only alternative is for them to implement the `Expression` and `BlockStatement` interfaces. However, removing `ExpressionBase` and `StatementBase` would result in too much repeated code across the whole grammar, so the grammar requires both the interfaces and the abstract classes. The enumeration `SingleWordStmnt` (discussed on page 16) must also be a MiniJava statement, and since enumerations cannot extend classes, it must implement `BlockStatement`, further cementing the need for both the interfaces and the abstract classes.

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 allows for the chaining of assignments found in the Java grammar [1, p.589]). 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 [1, 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, the expression A must only contain subexpressions that use operators with precedence at least as high as `+` and `-`, and B must only contain subexpressions that use operators with precedence higher than `+` and `-`. 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]

Figure 3.1 shows the top of the object hierarchy for the implementation of MiniJava. It demonstrates the distinction between MiniJava expressions and statements by way of the `Expression` and `BlockStatement` interfaces, the extra classification of expressions that can be used as statements given by `StatementExpression`, and shows the beginnings of the operator precedence hierarchy hardcoded into the expressions. The `MiniJASTNode` class is discussed in section 3.5.

Note also that as a side-effect of this implementation, the ambiguous code `1 + 2 + 3` also only has one possible interpretation, namely `(1 + 2) + 3`. This follows the established rules of operator associativity, which are now also hardcoded into the grammar.

In MiniJava, every expression has a type which consists of a primitive type and may or may not be an array type. There are four possible primitive types: `boolean`, `char`, `int`, and `double`. The primitive types are represented in the grammar by the enumeration,

[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.

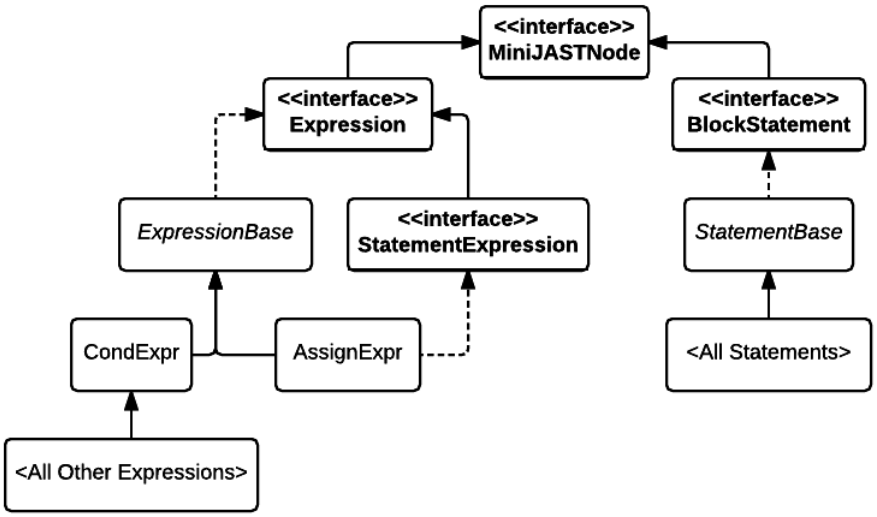

Figure 3.1: The UML diagram showing the top of the object hierarchy for `MiniJASTNode`s.

`PrimType`. The type of an expression then is represented by the class `Type`, an instance of which stores the appropriate PrimType value and a flag determining whether or not it is an array type. These `Type`s are use by the interpreter to determine whether or not variables can contain certain values, and whether operators are being supplied with the appropriate operands.

The majority of statements in MiniJava are composed of statements or expressions, but there are some that consist of only a single word, or even no words at all. To have an entire class representing each of these trivial statements seems wasteful, so the enumeration `SingleWordStmnt` represents four such statements: the break statement (`break;`), the continue statement (`continue;`), the return statement (`return;`), and the empty statement (`;`). Although there are no functions in the MiniJava grammar, the return statement can be used to immediately halt evaluation.

As we have seen previously, `AddExpr` defines the boolean field `isPlus`, to determine whether this instance represents an addition or a subtraction. There are some expressions where the number of possible operators that can be used in that expression is greater than two, in which case a boolean field will not be enough to disambiguate. `RelationExpr` and `AssignExpr` are the two expressions for which this is the case, since there are four relation operators and five assignment operators. These possible operators are represented by the two enumerations `RelationOp` and `AssignOp`. Thus `RelationExpr` stores a value of `RelationOp`, and `AssignExpr` stores a value of `AssignOp`.

There are two possible MiniJava expressions that can be assigned to, i.e. they can appear on the left hand side of an assignment expression. These two, identifiers and array accesses, are represented by the classes `Id` and `ArrayAccess`. To signify that they can both be assigned to, both classes implement the `AssignLHS` interface, an interface used for only this purpose. Similarly, there are two possible MiniJava expressions that can be assigned to an array type variable. These are array creation expressions (for example `new int[4]`) and array initialisation expressions (for example `{1, 2, 3, 4}`), which are represented by the classes `arrayCreation` and `ArrayInit` respectively. To signify that both can

be assigned to arrays, they both extend the abstract class, `ArrayAssignRightSide`. To justify why `AssignLHS` is an interface and `ArrayAssignRightSide` is an abstract class, we consider the difference between the expressions that can appear as subexpressions of other expressions, and those that cannot. Identifiers and array accesses may appear as a subexpression of some other expression. Because of this, the index operator used in array accesses must appear within the operator precedence hierarchy described previously, and so too must identifiers. If identifiers were not present in the operator precedence hierarchy, then they could never be substituted for the terminal symbols in the production rules seen previously, ruling out valid expressions like `i + j`. If identifiers and array accesses must appear in the operator precedence hierarchy, and high up in it too, then that means their representative classes must extend some class representing an operator lower down in the hierarchy. Thus `AssignLHS` must be an interface and not an abstract class, as `Id` and `ArrayAccess` already subclass some other class (namely `UnaryExpr`). Array creation and initialisation expressions however can never appear as a subexpression of some expression (since only single-dimensional arrays are allowed, expressions such as `{{1},{2,3,4}}` are not valid). This means that they do not appear in the operator precedence hierarchy, and thus their representations do not need to extend any base class. Thus `ArrayAssignRightSide` can usefully be an abstract class, reducing the amount of repeated code in these two classes.

MiniJava uses local variable declarations (for example, `int i = 4, ar[];`) in the same way that Java does: to declare and initialise new variables. Local variable declarations can be thought of as being made up of a primitive type and a list of so called "variable declarators", which are identifiers that may or may not be followed by square brackets to show the new variable is an array, and which may or may not be followed by an equals sign, "`=`", and a value with which to initialise the new variable. My implementation represents local variable declarations using the class `LocalVarDec`, which consists of a `PrimType` and a list of `VarDeclarator` instances. The `VarDeclarator` class stores a String containing the name of the variable, flags to determine whether the variable is an array type and whether an initialiser expression is given, and the initialiser expression itself if there is one.

What distinguishes MiniJava from other languages is that the concept of a blank, a missing piece of code, is built into the grammar itself. Representations of blanks in code should enable a user to fill one with their own code. To do this programmatically, it is necessary for representations of blanks to each come with a unique id, such that users can choose which blank they are filling and the blank be searchable within the AST representing the program. The abstract class `FillableBlank` fulfils this requirement by storing a `static` reference to an `AtomicInteger` and a private `int` field, `id`. Whenever a new blank needs to be represented in an AST, an instance of a class that extends `FillableBlank` can be supplied with a unique identifier that has not yet been used in this runtime environment, by using the `AtomicInteger`'s `incrementAndGet` method and storing the result in `id`. The two concrete classes that extend `FillableBlank` are `FillableBlankExpr` and `FillableBlankStmnt`. The first of these can be used to replace a MiniJava expression (and thus implements the `Expression` interface so that it can be stored in a MiniJava AST), and the second can replace a MiniJava statement (and thus implements

`BlockStatement`). `FillableBlankExpr` stores an `Expression` field, `studentExpr`, which, if not `null` is representative of the code with which a user has filled the blank. Similarly, `FillableBlankStmnt` stores an `BlockStatement` field, `studentStmnt`.

In this section I have described the implementation of my MiniJava language.

3.2 Interpreting the language

In this section I explain how MiniJava programs are interpreted.

To correctly interpret MiniJava, several requirements must be met by the interpreter:

1. The scope of variables must be handled, such that when a variable is used its current value can be accessed, and the same variable can't be declared twice in one scope.
2. Expressions must be able to propagate their values upwards, so that surrounding expressions and statements can use them.
3. Flow control must be handled, such that break and continue statements can be used in loops.
4. If an exception is raised during execution, the exception should propagate up and halt execution, preferably delivering a useful message.

The scope of variables is handled by the interpreter with instances of the `Context` class. Part of its definition is shown below:

```
public class Context {
    public Stack<HashMap<String, Type>> namesToTypes;
    public Stack<HashMap<String, Object>> namesToValues;
    ...
}
```

The `namesToTypes` field is a stack of maps from names (as `String` values) to instances of my `Type` class. Every variable that is declared in this scope will have an entry in the map on top of this stack, recording the type of that variable. The `namesToValues` field is a stack of maps from names to instances of `Object`. If a variable has an entry in the map on top of this stack, then the variable has the value stored in the map with it in this scope. It is possible for a variable to have an entry in the map on top of the `namesToTypes` stack, but not in the map on top of the `namesToValues` stack: this indicates that the variable has been declared but not initialised in this scope.

During execution, the interpreter must keep track when variables will go out of scope. This is handled by the `stepIn` and `stepOut` methods in `StatementBase`. The `stepIn` method is called whenever execution enters a new, deeper, scope. This method copies every key value pair in the maps on top of their respective stacks into new maps, and pushes these new maps onto the stack. Thus every entry in one of the stacks represents a different scope: as you move down the stack, you move outward through the nested scope. The `stepOut` method is called whenever execution leaves the current scope. At this point, all the variables that do not exist in the new scope should be forgotten, and

every variable that does exist in this new scope should have their value updated to the value it was in the old scope. The `stepOut` method first pops the top off both stacks, storing the top of the `namesToValues` stack in the local variable `oldMap`. Then, for every key in `oldMap`, if this key also appears in the new top of the `namesToTypes` stack, the corresponding value in `oldMap` is used to update the top of the `namesToValues` stack. In this way, variables can be updated in deeper levels of scope, and retain their new values when execution moves up the nested scope.

Consider the expression `(2*3)+4`. In MiniJava, this expression might be represented by a `AddExpr` storing a literal with the value `4` as the right operand, and an `MultExpr` as the left operand. This `MultExr` would store literals with values `2` and `3` as the left and right operands. During the evaluation of an expression, the most deeply nested parts of the expression must be evaluated first, so that their values can be used further up in the nested expression. There is no point trying to evaluate the `AddExpr` before we know the value of `(2*3)`. Thus, the `MultExpr` needs a way to communicate to the `AddExpr` that it has a value of `6`, so that the `AddExpr` can be evaluated to `10`. Likewise, whatever context the `AddExpr` is in needs to be told it has the value `10`. The way this is done in MiniJava is using implementations of the `ReturnValues` abstract class. This class represents the notion that some value is being returned by an expression, but allows any type of value to be returned, no matter what primitive type the value takes, and whether it's an array or not. To return a value of a particular primitive type, the appropriate one of these four implementations must be used: `ReturnValuesBool`, `ReturnValuesChar`, `ReturnValuesInt`, or `ReturnValuesDouble`. Each of these has a public `value` field of the appropriate type. If an array value is being returned by an expression, then a further implementation of `ReturnValues` must be used, the generic class `ReturnValuesArray<T>`. This class makes use of Java Generics as it stores an internal `ArrayList<T>` of values.

In the same way that MiniJava expressions dispense values, MiniJava statements dispense control flow commands. These commands can be to "break", to "continue", to "return", or there can be no command, in which case execution continues as normal. The MiniJava interpreter represents these commands using the `FlowControl` enumeration, which has the four values `BREAK`, `CONTINUE`, `RETURN`, and `NONE`. Whenever a statement is executed, the interpreter looks for which value is dispensed and reacts accordingly.

Several custom exceptions are used by the interpreter, to differentiate between when an error occurs in the MiniJava program being interpreted, and when an error occurs in the interpreter itself. All these custom exceptions subclass the base class of `MiniJavaException`, and the majority of them are analogous to standard Java exceptions. For example, the `OutOfBoundsException` is thrown by the interpreter whenever an index is used in an array where no such index exists. This exception is analogous to Java's `IndexOutOfBoundsException`.

The MiniJava interpreter is implemented using the `evaluate` method in the `Expression` interface, and the `execute` method in the `BlockStatement` interface. The method signatures of these two methods are shown below:

```
public interface Expression extends MiniJASTNode {
    ReturnValues evaluate(Context c) throws MiniJASTException;
    ...
}

public interface BlockStatement extends MiniJASTNode {
    FlowControl execute(Context c) throws MiniJASTException;
    ...
}
```

It can be seen that the interpreter makes use of all the components meeting the requirements for the interpreter. The `evaluate` method is overridden by every class representing a MiniJava expression, and the `execute` method is overridden by every class representing a MiniJava statement. This means that both methods are recursive: expressions being evaluated will call the `evaluate` method on their subexpressions for example. Typically, execution is initiated with an empty `Context` object that becomes more populated as more variables are declared. This interpreter can also be plugged in to peripherals. The values of variables in the outermost scope can be accessed after execution has terminated, and MiniJava has an inbuilt print statement, represented by the class `PrintStatement`, that can print the values of expressions to a file. Peripherals could also, if they wanted, make use of the custom exceptions thrown during execution, and the initial `Context` object.

It is important to remember that the MiniJava grammar also contains the notion of blanks in code, and thus the interpreter must be equipped to deal with these appropriately. If a blank has not yet been filled, then the interpreter should throw a custom exception, but if the blank has been filled, then execution must continue as if there was no blank there. The `FillableBlankExpr` and `FillableBlankStmnt` classes override the `evaluate` and `execute` methods in similar ways to fulfil these requirements. If the `Expression` or `BlockStatement` field representing student code is discovered to be `null` at evaluation or execution, then the custom exception `BlankEmptyException` is thrown. If, however, the field is not `null`, then the blank simply acts as a conduit to pass information to and from the student code. `FillableBlankExpr` evaluates the student expression with the `Context` object supplies, and simply passes the returned `ReturnValues` object upwards as if it were itself an evaluated expression, while `FillableBlankStmnt` does the same for statements except with the returned `FlowControl` object.

In this section I explained how MiniJava is interpreted.

3.3 Parsing the language

In this section I explain how a parser for MiniJava was implemented using the ANTLR parser generator [2], and how a representation of an input MiniJava program can be built using it.

The parser grammar (which can be found in Appendix A) is based on an existing ANTLR parser grammar for Java[2]. It would not have been worthwhile for me to write

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

an entire grammar from scratch when there was such a readily available alternative. It is always good to use existing libraries or tools if they are available.

The ANTLR tool generates recursive descent parsers from input grammars. The typical usage of ANTLR follows this process:

1. Write a grammar in a grammar file.
2. Run the ANTLR tool on this grammar file to generate a parser.
3. When this parser is executed with input source code, ANTLR's internal representation of the corresponding parse tree is built.
4. Using either a "listener" or a "visitor", walk the generated parse tree, taking actions at each node, processing the source code as desired.

The following code shows how to build a MiniJava representation for the source code `i = 4;`.

```
1 AssignExpr aE = new AssignExpr();
2 Id i = new Id();
3 i.setUpId("i");
4 Literal four = new Literal();
5 four.setUpLiteral(PrimType.INT, "4");
6 aE.setUpAssignExpr(i, AssignOp.EQ, four);
7 ExpressionStmnt eS = new ExpressionStmnt(aE);
```

Considering this representation as a parse tree, this can be seen as line 0 (i.e. the invisible line before line 1) visiting the `ExpressionStmnt` node and visiting its only child, the `AssignExpr` node in line 1. This node's first child is visited in lines 2 and 3, and its second child is visited in lines 4 and 5. Once all the children of the `AssignExpr` have been visited, the instance itself can be set up on line 6. Once all the `ExpressionStmnt`'s children have been visited, its instance can be set up on line 7. This example demonstrates that in general, program representations are generated by recursively visiting each expression or statement's children, in a similar way to how a program representation is interpreted.

By default, the ANTLR tool generates a listener interface and base class along with the parser. A listener is characterised by providing callback methods that are triggered by an automatic parse tree walker: listener methods don't have to explicitly visit their children. A visitor on the other hand gives the user more control over the walk, requiring explicit commands to visit the children of each node. While this means that it takes less work to use a listener (as the parse tree walker is already built in), the visitor paradigm is more appropriate for building my library's representation of the MiniJava source code. While the listener methods all return `void`, requiring the instances representing the children nodes to be stored somewhere in memory prior to setting up the current node, the base visitor, `MiniJavaBaseVisitor<T>`, is generic, with method signatures such as `public T visitBlock`. This means that the visitor methods can be made to return `MiniJASTNode`s, which can be cast to `Expression`s or `BlockStatement`s as appropriate (as shown in Figure 3.1 on page 16). By using a visitor rather than a listener, I was able to more closely

imitate the way (shown above) that program representations are build in code. My visitor implementation can be found in `MiniJavaASTBuilder`, a class which extends the `MiniJavaBaseVisitor<MiniJASTNode>` class supplied by the ANTLR tool.

Java, and by extension MiniJava, is a language that suffers from the "dangling-else" problem: a particular problem arising from an ambiguous grammar. Consider the MiniJava code shown below:

```
if (i == true)          (1)
    if (j == false)     (2)
        k = 1;
else
    k = 2;
```

A parser could not make a decision as to whether the `else` statement "belongs" to the first or the second `if` statement. Since the grammar is ambiguous, a decision must be made for the parser. The decision taken in Java[1, p.417], and thus in MiniJava, is to attach the `else` statement to most inner `if` statement possible (which in the example above would be the one labeled `(2)`, even though the indentation may suggest that the programmer intended it to be the one labeled `(1)`). To implement this decision, the grammar distinguishes between `Statement`s and `StatementNSI`s (where NSI stands for "No Short If", i.e. no `if` without `else`). The way this solves the problem becomes apparent on inspection of the following rules in the grammar (presented in context in Appendix A):

```
statement:
    ...
    | IF parExpression statement
    | IF parExpression statementNSI ELSE statement
    ...
    ;

statementNSI:
    ...
    | IF parExpression statementNSI ELSE statementNSI
    ...
    ;
```

The `statement` rule gives productions for both a short `if` (i.e. with no attached `else`) and a long `if`, while the `statementNSI` rule only produces a long `if`. It can be seen that using these rules to parse the code above will reach a dead end if the first `if` statement is parsed as a long `if`, since its first sub-statement would have to be a `statementNSI`, and this rule has no viable way of parsing the second `if` statement. Of course, the visitor rules for `statement`s and `statementNSI`s make no distinction as they build the MiniJava representation, returning just a standard short or long `if` representation as appropriate.

Another problem encountered while writing the parser can be discovered on inspection of the local variable declaration production shown below:

```
blockStatement:
    primitiveType variableDeclarators SEMI    # localVariableDeclaration
    ...
    ;
```

This production is used in the creation of `LocalVarDec` instances, which require the appropriate `PrimType` to set up correctly. As the visitor arrives at a node produced like this in the parse tree (by calling the `visitLocalVariableDeclaration` method found in `MiniJavaASTBuilder`) it needs to visit the node's children in order to gather the necessary objects to set up the `LocalVarDec` object. Since visitor methods return `MiniJASTNode`s, the `visitPrimitiveType` method (also found in `MiniJavaASTBuilder`) needs to return an object whose class implements `MiniJASTNode` and represents a primitive type. Until this point, no such class existed. The enumeration `PrimType` does not implement `MiniJASTNode`, as that would require overriding all the methods declared in the interface separately for each value, as they are in `SingleWordStmnt`. Since this would be a lot of wasted code, I instead wrote a new class, `PrimTypeCarrier`, for precisely this purpose. This class implements `MiniJASTNode` and has a public field of type `PrimType`, meaning it only needs to implement (trivially) all of the interface methods once. An instance of this class can thus be created in the `visitPrimitiveType` method (setting the `PrimType` field value appropriately), which can then be returned to the `visitLocalVariableDeclaration` method for setting up the `LocalVarDec` object.

In this section I have described how the ANTLR parser generator can be used to generate parsers for the MiniJava grammar, and how program representations can be built using visitors.

3.4 Representing questions and solutions

In this section I explain how the general notion of a question is represented using the `AbstractPExercise` class, and give an example of how specific questions can be encoded by extending this class.

When presented to a student learning a programming language, a "question" can be broken up into the following components:

- A description of what the student is expected to do.
- A model solution to the problem presented with parts of the code left blank for students to fill in.
- A measure of difficulty so that both teaching staff and students can monitor progress and assess performance.

Following these guidelines, a representation of a programming question should be capable of the following:

1. Storing the description of what the student is expected to do.
2. Building a MiniJava representation of the model solution.

3. Adding and removing blanks from the solution to make the problem harder or easier.
4. Calculating the difficulty of the current problem, given how difficult the question is to answer and how much of the model solution is blank.
5. Filling those blanks with MiniJava representations of student submitted code.
6. Executing the solution.
7. Checking that the problem is solved by the provided solution.

The abstract class `AbstractPExercise` provides the framework for classes extending it to fulfil all these requirements. I explain how the points above are addressed by `AbstractPExercise` in order.

1. Storing the description of what the student is expected to do

The `String` field `question` stores the question provided to students. This field is initialised in the constructor. In order to present the model solution along with the question, MiniJava program representations need to be converted back in to `String` representations. To implement this, the `stringRepr` method was added to both the `Expression` and `BlockStatement` interfaces, with the latter's version taking an integer argument representing how many blocks deep the statement is. This method is implemented in every MiniJava expression and statement by recursively combining String representations of the sub-expressions and sub-statements, effectively turning the representation in the abstract syntax into one in the concrete syntax. Depending on how blanks are filled by the specific peripheral in use, it may be useful to print blanks with their unique ids or without. The `FillableBlank` base class records with a `static final boolean` flag whether ids should be shown or not. If they should, then the blank with id n is printed: "`..(n)..`", otherwise it is printed "`.....`".

2. Building a MiniJava representation of the solution

Since the solution to the exercise needs to be executed (to ensure that it is correct), it must be stored as a representation of the MiniJava code. This representation is built by the parser using the `String` field `solutionCode` as input. The representation is stored in the `BlockStatement` field `solution` of the `AbstractPExercise` class.

3. Adding and removing blanks from the solution

The algorithms used to add and remove blanks from the solution code are complex enough to warrant their own section, and are thus discussed in section 3.5.

4. Calculating the difficulty of the current problem

As already stated, the difficulty of a question is related to how challenging it is to answer the question, and how much of the model solution is blank. Research into pedagogy was beyond the scope of this project, and it is expected that teaching staff would have a better

understanding of how difficult different exercises would be for their own students anyway, so for now the difficulty measurement system is very much a place-holder. An integer is supplied to the base constructor by every implementation indicating the supposed difficulty of that question, with higher numbers assumed to indicate harder questions. This integer is combined with a float measuring the percentage of the model solution that is currently blank to give a measure of the difficulty of the overall problem. This means that for a question with difficulty n, the difficulty of the associated problem can range from n to $n + 1$.

To work out how much of a MiniJava AST is blank, we need to count how many nodes in this tree have been replaced by a blank, and divide that by the total number of nodes in the tree. In order to calculate how many nodes in a tree have been replaced with a blank, we need to know how many nodes have been replaced by each *specific* blank. As such, the `FillableBlank` base class stores an integer, supplied at a blank's construction, that records how many nodes this blank replaced. The standard way to count the number of nodes in a tree which has root N is to count the number of nodes in each subtree that has a root that is one of N's children, and add one to the total. Since we are counting not only the total number of nodes in the tree, but also how many of those nodes are blank, the method used to count the nodes must return both these numbers simultaneously. This is done using the class `NodeCount`, which stores two integer fields, one, `empty`, records the number of blank nodes in the tree, and the other, `filled`, records the number of nodes which are not blank. The `getTreeSize` method returns a `NodeCount` object, and is implemented by every class in the MiniJava implementation. It uses the recursive approach to counting nodes in a tree, distinguishing between nodes which are blank and nodes which aren't. To work out how much of the AST is blank, this method is called on the root of the tree so that the returned `NodeCount` object can be used to calculate the percentage using the formula `empty / (empty + filled)`.

5. Filling blanks with student submitted code

Since representations of blanks in code have their own unique identifier, the blank that a user wishes to fill can be selected using it. To fill a blank with the representation of a user's code, the representation of the blank itself must be found within the program's AST, before the representation of the code is stored within it. This functionality is found within the `fillBlank` method of the `AbstractPExcercise` class. This method takes the identifier of the blank to be filled and the AST of the code to fill it with. The method then uses a depth-first-search through the AST to find blanks which, when found, have their identifiers compared with the target identifier. If they do not match, then the search moves on, but if they do match, then it is determined whether the blank is an instance of `FillableBlankExpr` or `FillableBlankStmnt`. If it is the former, then the blanks `studentExpr` field is set to the user's AST. If it is the latter, then it is the `studentStmnt` field that is set. If the search fails, then it can be reported that no blank with the suggested identifier exists within the solution.

6. Executing the solution

Since the `AbstractPExercise` class stores the AST of the solution, executing the solution is as simple as using the inbuilt interpreter on it. The `runSolution` method in this class initialises its `c` field, which is a `Context` object for use in execution and solution verification, and calls the solution's `execute` method with `c` as an argument.

7. Checking that the problem is solved by the provided solution

Since different questions have different requirements, the method in `AbstractPExercise` that checks the validity of solutions, `checkSolved`, must also be abstract. This method returns a `boolean`, which will have the value `true` if the solution is correct and `false` otherwise. There are two possible ways of checking whether a solution is valid: either the value of an variable can be compared with a target value, or the result of a print statement can be inspected. The first of these approaches requires an `Id` field to be declared in the concrete class representing the problem, assigned the appropriate name in the constructor. In the overridden `checkSolved` method, the final value of this variable can be accessed by extracting the `value` field from the `ReturnValues` object attained by calling the `Id` instance's `evaluate` method with `c`, the `Context` object used during execution. In contrast, the second approach can be implemented by checking the contents of the file written to by the representation of the print statement.

An example concrete class extending `AbstractPExercise` is `FactorialExercise`, which represents 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, storing the result in the variable `total`. The model solution code is shown below:

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

This code calculates $N \times (N-1) \times \cdots \times 2 \times 1 = N!$ and stores it in `total` as required. In the `FactorialExercise` constructor, the value of $N!$ is calculated in the same way and stored in the `int` field, `nFact`, and the `totalID` field of type `Id` is set up with the name "`total`". The overridden `checkSolved` method evaluates `totalID` with the `Context` object `c` and compares its final value with the `nFact` field, returning `true` if they are the same and `false` otherwise.

In this section I have presented the class `AbstractPExercise`, shown how it fulfils the requirements for representing an exercise, and shown a concrete example that extends it.

3.5 Adding and removing blanks

This section describes 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.

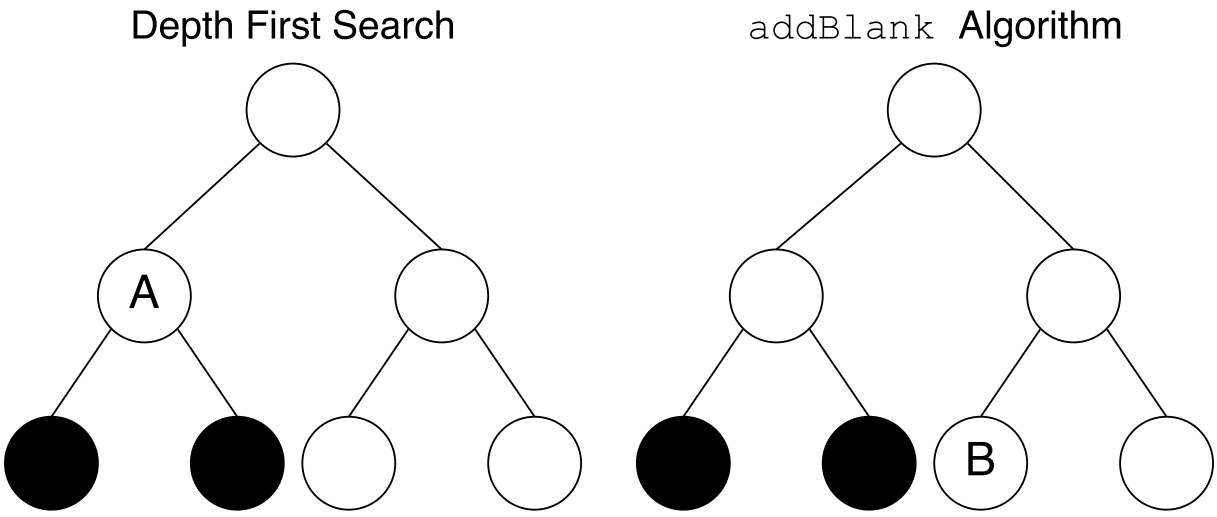

Figure 3.2: A figure demonstrating the different behaviour of a depth first search algorithm and my `addBlank` algorithm.

The purpose of these algorithms is to offer the most fine-grained adjustment of the number of blanks in a model solution possible, which in turn will give the greatest possible degree of control over the difficulty of the problem. This means that the algorithm that adds blanks to a model solution must select the blank at each iteration of the algorithm which causes the smallest increase possible in the percentage of the solution which is blank. In order to ensure this, the algorithm always selects the left-most, deepest 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 on the third iteration by a depth first search algorithm and my algorithm. 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 we should select node B, as it is the next deepest leaf. 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 we switch which marking we are looking to replace, and node A will be made blank.

For this algorithm

This section described the implementation of the algorithms that introduce blanks 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.

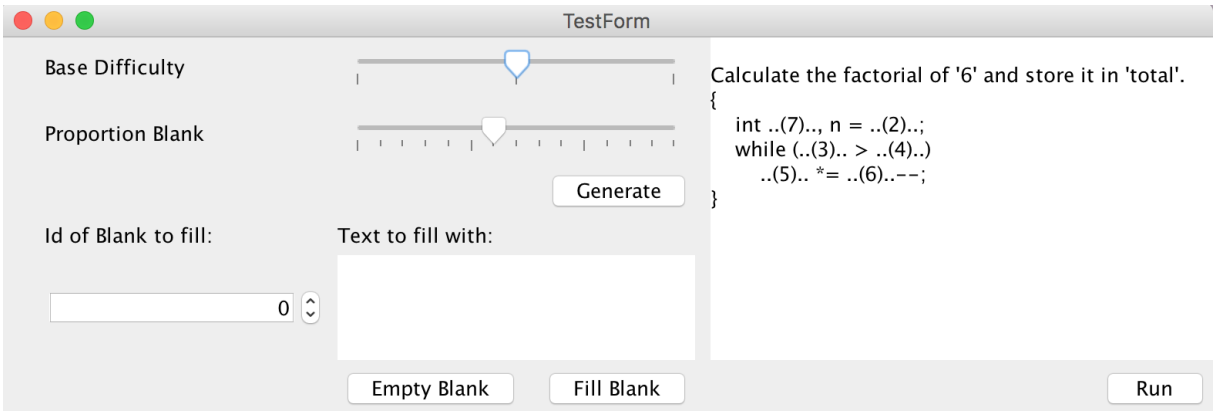

Figure 3.3: An example screenshot taken of the GUI peripheral.

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.

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

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

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

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

[1] J. Gosling, B. Joy, G.L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Java (Addison-Wesley). Addison Wesley Professional, 2014.

[2] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

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:

1. I have a system that presents the same initial problem to all students.
2. The system can determine that code written by the student solves the given problem.

3. On registering completion of the exercise, the system will then present a new problem to the student.
4. If the student solved this problem with sufficient ease as measured by my performance heuristic, then this new problem will be more difficult, as measured by my difficulty heuristic.
5. If the student failed to solve the problem, or took too much time or made too many errors, then the next problem will be easier.
6. The system will adjust the difficulty of a given problem in line with the difficulty heuristic by changing the amount of code required to solve the problem, and also the underlying problem being solved.

Possible extensions

- This system could be improved such that it identifies which concepts a student is struggling with and adjusts the content of future exercises to encourage learning of these concepts.
- The program could produce a graph of progress against time to motivate learners.
- A simple game that utilises this system could be made, to showcase how the system might be used to teach programming.

Timetable

24th Oct - 6th Nov:

- Research adaptive difficulty and see how it has been done before.
- Research how errors in code have been quantified in the past.
- Research where and how simulated users have been designed to test systems.

7th Nov - 20th Nov:

- Describe how code that solves an exercise could be split up into discrete sections, and a way to store how well each of those sections was completed.

Milestones:

- Have a representation of a programming puzzle in Java code.

21st Nov - 4th Dec:

- Prototype and compare heuristics to measure the "difficulty" of a given programming exercise, based on the difficulty of the goal to be achieved, and how much of the code is presented initially.

Milestones:

- Select and implement a difficulty heuristic.

5th Dec - 18th Dec:

- Prototype and compare heuristics to measure performance, based perhaps on how much time was taken, the number of errors made and the time the solution takes to run.

Milestones:

- Select and implement a performance heuristic.

19th Dec - 1st Jan:

- Slack time over Christmas to catch up if I need to.

2nd Jan - 15th Jan:

- Implement a system to adapt the difficulty of the programming exercises by presenting a new, more difficult problem each time an exercise is completed satisfactorily, and presenting an easier one if the student is struggling too much.

Milestones:

- Have a system that will present a problem to a student, determine that the student has solved the problem with the code they have written, determine if the student found it too easy or too hard, and present the student with a new problem that is either easier or harder respectively.

16th Jan - 29th Jan:

- Write Progress Report.
- Improve the system by implementing the concept of changing the amount of code that needs to be written to solve a given problem as another way of affecting its difficulty.

Milestones:

- Have the system extended such that it may require the student to write more or less code to make more fine grained adjustments to the difficulty of a programming problem.

30th Jan - 19th Feb:

- **3rd Feb:** Submit the Progress Report
- Design a user interface to change the content of the programming exercise to be solved.

Milestones:

- Have a user interface with controls to determine the exact difficulty of the problem about to be presented.

20th Feb - 5th Mar:

- Write unit tests for all units.

Milestones:

- Have unit tests written for every unit of the code.

6th Mar - 19th Mar:

- Test the implementation by simulating a student "using" the system by "solving" problems with varying success.

Milestones:

- Have a simulated student that will tell the system what mistakes were made in the code and where, as well as how long it took to solve the problem (along with any other parameters deemed important to measuring the performance of the student), such that the response of my system can be measured against the performance of a student using it.

20th Mar - 2nd Apr:

- Slack time for catching up and doing extensions.

3rd Apr - 16th Apr:

- Write Introduction and Conclusion (about 2,500 words).
- Write up Proforma (excluding word count), Declaration of Originality, Project Proposal and Cover Sheet.

Milestones:

- Have the above sections of the dissertation written in draft form.

17th Apr - 30th Apr

- Write Preparation and Implementation (about 5000 words).

Milestones:

- Have the above sections of the dissertation written in draft form.

1st May - 14th May

- Write Evaluation (about 2500 words).
- Write Contents Page, finish Bibliography, Appendices and Index as required.

- Fill in word count of Proforma.
- Submit draft to DoS and Supervisor.

Milestones:

- Have the above sections of the dissertation written in draft form.

15th May - 19th May

- Incorporate feedback into dissertation.
- **19th May:** Submit dissertation.