Computer Science AP/X
Augusta the Robot

CSCI-140/242
Project 2

11/28/2016

Due Date: Wednesday, 30 November 2016, 11:59pm

1 Goal

Build a visual code development tool and visual simulator for a simple programming language that controls a robot, named "Augusta", that moves around on a grid with blocks and crumbs.

2 Team Work

This is an ambitious project that requires a significant amount of time for planning, design, and implementation. It has been laid out to allow two persons to work fairly independently (if they want) to build two separate components that will, if they follow this specification, work together.

You are expected to work on this project with a partner. If you do not choose a partner for yourself, one will be chosen for you. The instructors reserve the right to deny a specific student pairing if they feel that a pairing would not be in the best interests of the students involved.

Each student in the team is expected to be the *lead*, and therfore take primary responsibility, for one of the two major system components.

3 Skills Demonstrated

This is an ambitious project that requires many skills.

- Handling tree structures
- User interface design
- Parsing
- Grid-based "game" animation
- Larger system design
- Team work

4 Overview

Manipulation of Augusta and the world with which she interacts requires a programming language to send her instructions. You will be using a custom language whose semantics, but not necessarily syntax, is specified in this document.

This system has a *front end* that is involved with getting the user to write a program in Augusta's language. It should be a graphical user interface, but other than that, the apparent structure of the programs a user writes is up to you. Once the user is happy with a program, the front end either stores the program in a file or passes the compiled program to the back end.

The notion of visual programming has been around for a while, and is a popular approach to help beginners, especially children, learn how to do compuational design. For widely-used systems that use this approach (but are far more sophisticated), see

- MIT App Inventor, appinventor.mit.edu
- Scratch, the basic language from MIT, scratch.mit.edu
- Introductory levels of Alice, from CMU, at alice.org

Figure 1 is a (crude) image from the on-line Scratch development environment.

Figure 1: Screen Shot of Scratch Development Environment

The system's *back end* is concerned with running and displaying a simulated environment in which Augusta executes the user's program. Again, the exact manner of visualization is up to the students, but since the robot's world is described as a grid-based rectilinear environment, one would expect to see that manifest in the display.

Figure 2 is an image from a similar kind of simulation.[1]

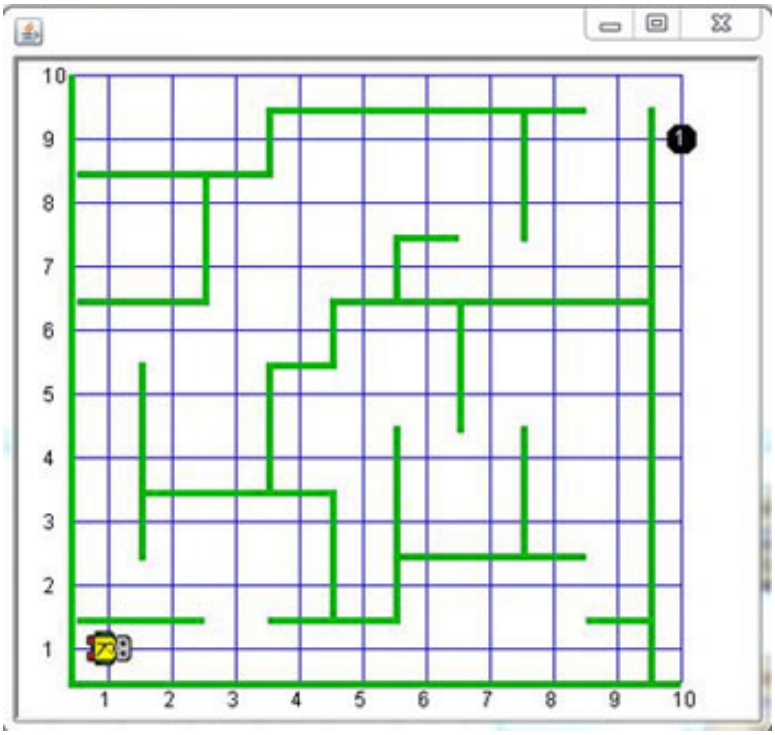

Figure 2: Another Robot-In-A-Grid Application. This one uses walls, not blocks.

In between there is the programming language's intermediate form, which is provided to you in code as a set of tree nodes that the front end component can use to build an *abstract syntax tree*, and that the back end interprets, along with an initial world configuration file, to effect a simulation of Augusta and her world.

Augusta moves around in a world of cells arranged in a Cartesian fashion – a *grid*. Augusta occupies one specific cell at any point in time. She has a *heading* in one of the four standard compass directions. If she moves, she moves one cell along her current heading.

Some cells are occupied by blocks. Augusta cannot occupy a "blocked" cell. If her instructions command her to do so, an exception is raised. Similarly, she cannot exit the bounds of the world.

Some cells are occupied by *crumbs* in Augusta programs. They can be used as markers. Augusta can put a crumb down in a cell, check if one is in her cell, and pick one up. Attempting to pick up a crumb where one does not exist again raises an exception. (However,

1. Do not consider this image to reflect a project specification. There are some basic differences.

putting one down where there is already a crumb does not. In fact, it does not change the state of the world at all. Crumbs do not accumulate.)

5 Provided Design and Software

An *abstract syntax tree* (AST) is a structure commonly used to represent programs that have undergone the early phases of compilation so that they are no longer linear sequences of text. They instead represent the recursive structures inherent in programming languages. Here are some examples from Java of the recursive nature of programming languages.

- A program is a set of classes.
- A class is a set of features.
- Features can be fields or methods.
- A method is a return type, list of parameter types, and a statement block.
- There are many kinds of statements, including block, if, and while.
- A type is a class or primitive type.
- A class is a regular class, array, interface, enum, or annotation.
- An if statement is a condition and two statement blocks.
- A while loop is a condition and a statement block.
- A statement block is a sequence of statements.
- A statement contains expressions.
- An expression can be built from simpler expressions, constants, variables, and lambda expressions.
- A lambda expression is a list of parameter types and a statement block.

Note how these definitions depend on each other and can even have cyclic dependencies. Mathematicians, programming language computer scientists, and linguists call this kind of structure a *context-free language*.

Trees are a natural way to express these kinds of structures. Your system will allow a user to "write" a largely linear program from a much smaller, simpler set of statements in terms of graphical objects on the screen. the system then "compiles" that information into an AST.

For this project, however, we have simplified the structure so that the program is represented by a sequence of ASTs, which is sometimes called a *forest*. Each tree's root represents a top-level statement in the program. Sub-blocks for a conditional or loop statement would appear further down in that statement's tree. This will be made clear once enough of the design has been explained to allow an example to be shown.

The tree node types we provide have been designated as `Serializable`. This means they can be written to instances of `java.io.ObjectOutputStream` and read from instances of `java.io.ObjectInputStream`. This is how you accomplish the requirement that a tree can be stored in a file by the front end and then read in and executed by the back end.

The back end tells each tree in turn to execute itself. The code we give you implements this by calling methods in the simulated world that Augusta lives in.

You are given a package called `augusta.tree`, whose UML class diagram is shown in Figure 3 that contains node types for all the statements your project must support. Certain additional types are added for uniformity of designs: `enum` types and an exception type.

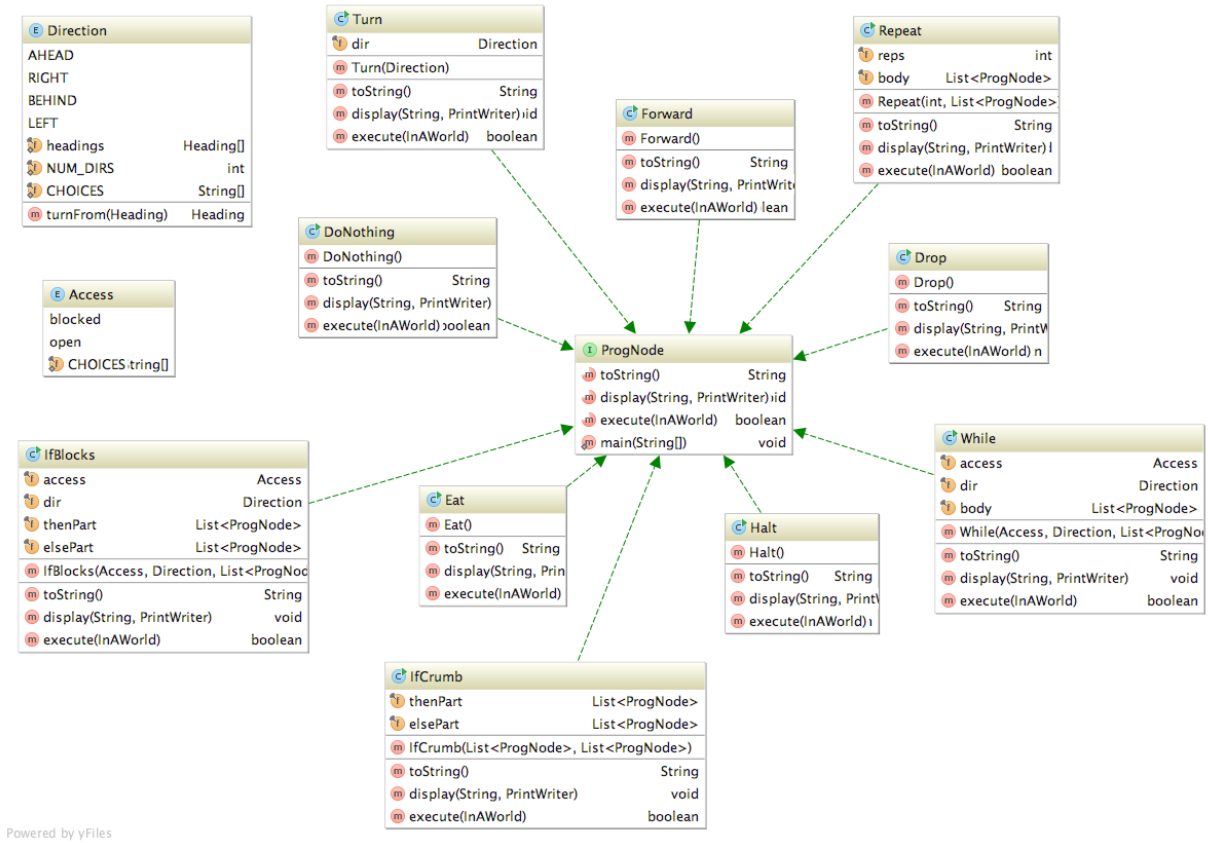

Figure 3: The `augusta.tree` Package Class Hierarchy

The robot commands are given in Table 1.

5.1 Summary

In this section the functionality of each of the two halves of this product is summarized. Details on each half follow in later section. However, keep in mind that you have significant latitude as far as the user interface design is concerned.

One member of the team is expected to do the front end "developer's tool". This tool will allow a user to visually compose a program for Augusta. Functionally this program builds a sequence of `ProgNode` trees. Once the user decides the program is ready, the developer's tool opens a file, of the user's choosing, as an `ObjectOutputStream` and iteratively writes to it the sequence of `ProgNode` trees.

The other member of the team is expected to do the back end "interpreter tool". This tool starts by reading in the aforementioned file, *plus* another text file containing the configuration

Command Name	Parameters, if any	Description
Forward		Augusta moves forward one square in her current heading. If blocked or at world's end, an `ImpossibleMove` exception is thrown.
Drop crumb		Augusta places a crumb in the current square. No effect if there is already a crumb there.
Eat		Augusta eats the crumb in the current square. If no crumb, an `ImpossibleMove` exception is thrown.
Turn	Direction d	Augusta turns. (The parameter value d is relative.)
IfCrumb		If a crumb is present on current square then Augusta executes the *thenPart* children of this node; else Augusta executes the *elsePart* children of this node.
IfBlocks	Access a, Direction d	If the d side of Augusta is a (open/blocked) then Augusta executes the *thenPart* children of this node; else Augusta executes the *elsePart* children of this node.
Repeat	integer n	Augusta executes the children of this node n times. n is limited to a positive integer less than 10.
While	Access a, Direction d	Augusta executes the children of this node as long as the d side of Augusta is a (open/blocked).
Halt		Augusta stops and the simulation ends.
DoNothing		Used as a filler child for the unneeded option in an IfCrumb or IfBlocks node.

Table 1: Augusta's Command Semantics

of Augusta's *world*. The world is graphically displayed, and Augusta interprets and executes the nodes of the program.

Both parts are expected to follow the Model-View-Controller architecture pattern. Also for both parts, once the graphical interface is displayed, all interaction with the tools must be there, and not in the console. The likely consequence of this rule is that the only thing that can happen in the console is for the interpreter to report that it cannot open the files it was given on the command line. (See **Interpreter Tool** below.)

6 The Two Components

In this section more detail is provided on the two parts of the program.

You have the choice of developing two separate GUI applications, thus forcing the user to save the program as a file, start the interpreter, and read the file back in. However, it is also acceptable to combine the two ends, and let the developer tool send the program forest directly to the interpreter. But in that alternate configuration it should still be an option to save and load program files.

If you write two programs, the main classes should be called `AugustaDeveloper` and `AugustaInterpreter`. If you submit a single program, its main class should be `Augusta`. These classes should be in the package named `augusta`. No classes in the default package are allowed.

6.1 Developer Tool

It is required that the user be presented with a visual choice, a *palette*, if you will, of program elements to use. The user should then be able to copy, without typing anything, those elements into a vertically organized program of some kind in another area (we'll call it the *development area*) of the GUI.

The vertical organization implies the order in which the elements will execute.

Clearly every `ProgNode` type (see Figure 3) should have some kind of exemplar or prototype in the palette. You may add others for code organization. (There is a recommendation about this later in the document.)

You must also have a way of modifying the parameter values for statements that have parameters, without use of the console.

There should also be a method for saving the program in a file. Recall that the format of the file is binary. It is the serialization of the entire sequence of `ProgNode` trees you somehow generate from the code in the visual development area.

Note that saving the code in this form makes it difficult to read it back in to the developer component for revisions. The ability to do so is *not* a requirement of this project.

You will need to include a short text document **instructions.txt** with your project submission that explains the following.

- how to copy program elements from the palette to the development area
- how to change the parameter values for those elements that have parameters
  - `IfBlocks`
  - `Repeat`
  - `Turn`
  - `While`
- the use of any program elements that do not correspond to a `ProgNode` implementing class

6.1.1 Design Tips for the Developer Tool

If you associate each code statement in the development area with its position on the screen then it is fairly simple to make a list of statements in the correct order: Just sort the statements by their y coordinates.

More sophisticated visual programming tools show subordinate statements inside a flow statement like an *if* or a *loop* by showing them being half swallowed by the flow statement. The latter partially surrounds the former. (See Figure 1.) This appearance is well beyond

what we expect of you in this project. Our suggestion is to instead add `begin` and `end` statements to the palette to do the grouping. In Figure 4 you can see a crude, but definitely acceptable way of utilizing grouping statements so that it is clear that the `FORWARD` and `IF_CRUMB` statements are part of the loop. The indentation does not need to be enforced, either.

Use file chooser dialogs (package `javafx.stage`) for writing files.

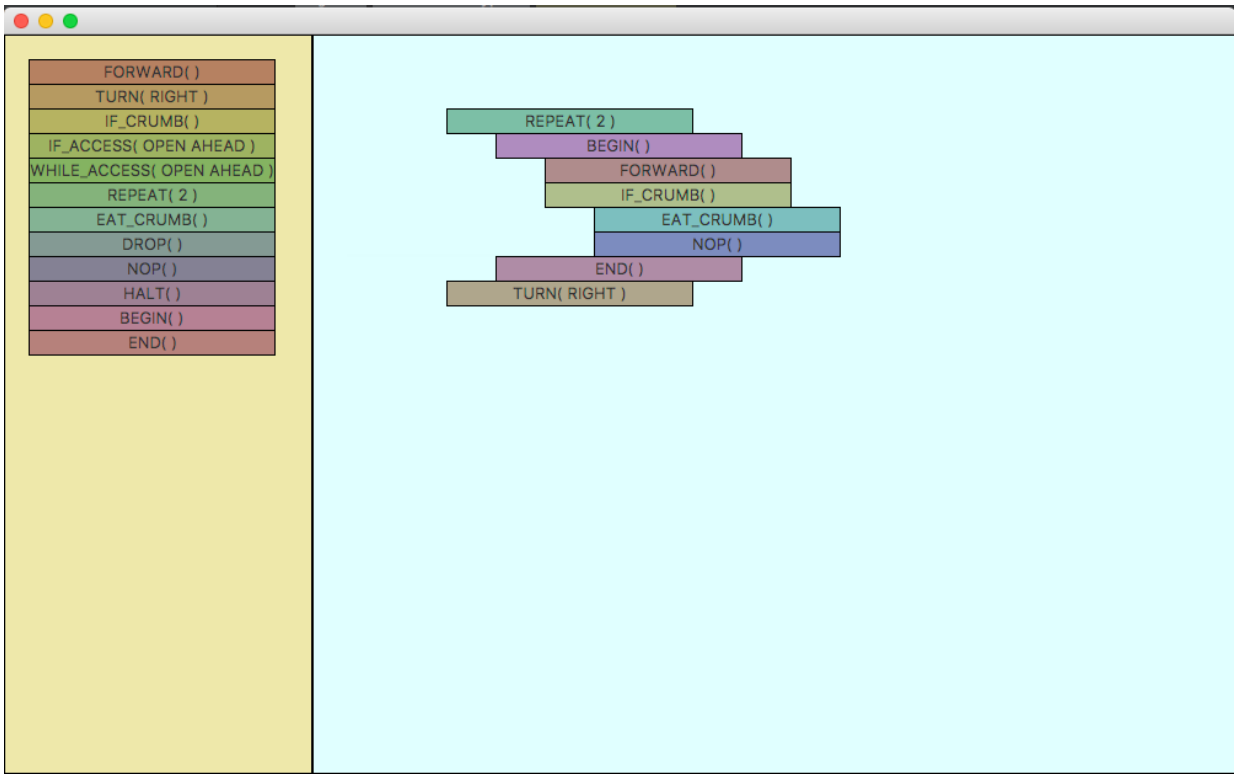

Figure 4: Augusta code Serving Suggestion

You should also check out the drag-and-drop code example that is posted for the weeks when GUI programming was covered. Be sure to check out the Ensemble mega-example for other potential ideas and code.

6.2 Interpreter Tool

The interpreter tool has to read two files in order to run.

1. the binary file containing the program, saved by a developer tool
2. a text file describing the world in which Augusta will execute the program

The names of these files must be specifiable on the command line.[2] Dialog boxes to choose alternates after the interpreter starts running are optional.

2. This is mostly to speed up testing.

6.2.1 World File Format

Here is sample world file.

```
# open+crumbs

6 6
B B B B B B
B . - . - B
B - . . . B
B . - - . B
B - . . . B
B B B B B B
1 1 EAST
```

After skipping initial lines that are empty or begin with the comment character "`#`", you read the configuration as follows.

1. the total size of the world as two integers: number of rows(r), number of columns(c)
2. r lines, each designating the contents of the cells on that row:
   - `B` a block, designating a cell where Augusta cannot go
   - `-` an empty cell
   - `.` a pre-existing crumb in the cell
3. the initial setup for Augusta: row position, column position, and heading (0-based)

Any lines after the last one described should not be read.

You may assume that these world files are error-free.

As mentioned earlier all classes that implement `ProgNode`, i.e., all nodes of trees in the program forest, implement this method:

```
execute( InAWorld w )
```

So at run-time the interpreter calls `execute(`*`world`*`)` on the root of each tree in the top-level `ProgNode` sequence. Calling of `execute` in non-root nodes is handled internally by the tree itself. However almost all implementations of `execute` in the various implementers of `ProgNode` call on the interpreter's world object to find out what's going on and to make changes to that world. This is where you come in. You must write the `InAWorld` implementing class.

The graphical tool of the interpreter back end should clearly display all the world information, and how it is changing. This is done by always showing a grid of cells, and who or what is on each cell. See the separate "tips" document to explain how to maintain an updated display after a single "RUN" button-press event.

The tool should at minimum have these function features.

- Load an Augusta program from the file name provided as the first command-line argument.
- Load a world file from the file name provided as the second command-line argument.
- Run (or re-initialize and rerun) the program.

6.2.2 Design Tips for the Interpreter Tool

Event handlers are supposed to be fast, so as to not cause delayed response when the user does something else to the GUI. (There is one thread that checks for events and runs event handlers. When your event handler is running, no events are being found.) In truth, running an entire Augusta program simulation is too long. Some code suggestions have been made in the separate "tips" document about how to create a separate thread of execution for Augusta so the rest of the system can keep running.

Many students in the past have used buttons as grid cells to avoid learning how to do a pure graphical grid layout. (Your partner on the front end is likely not to have that choice!) When you push the buttons nothing happens, but you can show a state change in its cell by swapping a new image (`ImageView`) onto the button. Another approach is to build a `Pane` or `Rectangle` with an image in it. Swapping images could be done by having a `StackPane` and putting a different image on the top at different times.

Be sure to check out the Ensemble mega-example for other ideas and code.

7 Provided Code

For all provided project files, including this document, go to
https://www.cs.rit.edu/ csapx/Projects/Augusta/.

You can download a jar file from there that contains the following classes.

- package `augusta.interpreter`
  - `InAWorld` interface
  - `ImpossibleMove` exception
- package `augusta.properties`
  - `Access` enumeration
  - `Direction` enumeration
  - `Heading` enumeration
- package `augusta.tree`
  - `ProgNode` interface
  - `DoNothing` class
  - `Drop` class
  - `Eat` class
  - `Forward` class
  - `Halt` class
  - `IfBlocks` class
  - `IfCrumb` class
  - `Repeat` class
  - `Turn` class
  - `While` class
- some potentially handy items in a package called `rit.edu.cs`

You should add the jar file as a library to your project.

Keep in mind that it is a *requirement* to represent the program as a sequence of `ProgNode` objects. We want to make sure to maintain compatibility between all the front ends and all the back ends. For the same reason, do not modify this code. Exceptions may be granted, but you should talk to your instructor about it.

There are two additional *executable* jar files. They are crude console applications that let you build and run Augusta programs. Below you will find a sample run, in a bash shell, of the use of these aids.

```
$ java -jar developer_ptui.jar
Augusta> add FORWARD
   0: FORWARD(  )

Augusta> add HALT
   0: FORWARD(  )
   1: HALT(  )

Augusta> store sill.aug
The syntax tree resulting from parsing:
Forward
halt

Augusta> quit

$ cat world_with_exit.txt
# open+crumbs

6 6
B B B B B B
B . - . - -
B - . . . B
B . - - . B
B - . . . B
B B B B B B
1 1 EAST

$ java -jar interpreter_ptui.jar sill.aug world_with_exit.txt
 XXXXX XXXXX XXXXX XXXXX XXXXX XXXXX
 XXXXX [ * ] [   ] [ * ] [   ] [   ]
 XXXXX [   ] [ * ] [ * ] [ * ] XXXXX
 XXXXX [ * ] [   ] [   ] [ * ] XXXXX
 XXXXX [   ] [ * ] [ * ] [ * ] XXXXX
 XXXXX XXXXX XXXXX XXXXX XXXXX XXXXX

Augusta is at (1,1) facing EAST.
```

```
Reading program..
Forward
halt

 XXXXX XXXXX XXXXX XXXXX XXXXX XXXXX
 XXXXX [ * ] [   ] [ * ] [   ] [   ]
 XXXXX [   ] [ * ] [ * ] [ * ] XXXXX
 XXXXX [ * ] [   ] [   ] [ * ] XXXXX
 XXXXX [   ] [ * ] [ * ] [ * ] XXXXX
 XXXXX XXXXX XXXXX XXXXX XXXXX XXXXX

Augusta is at (1,2) facing EAST.
$
```

Finally, a Java source file containing a "decompiler" is included. You can use it to verify that your developer program writes the code file correctly, as well as look at the contents of others' code files.

8 Grading

Each person in the team will assume primary responsibility for one component of the project. That person's name should be listed as first author on every file submitted for his/her component of the project.

Whether or not you materially contributed to the other component, the grade on that component will affect your grade as well.

- 80% of your grade is the grade assigned to your primary component.
- 20% of your grade is the grade assigned to your teammate's primary component.

The grade breakdown for each component in this assignment is as follows:

- Functionality: 65%
  - File Input/Output: 15%
  - Tool manipulations' conformance to specifications: 40%
  - Error indications: 10%
- Design: 25%.
- Code Documentation and Version Control: 10%

Extra credit may be given to teams that implement additional functionality. The usual rule about not obscuring/removing required functionality applies.

9 Submission

You will be using a team account for your project development and for submission. Only one member of each team needs to submit the project. We will not grade multiple submissions.

If more than one submission is made by one student, the most recent one will be graded. If submissions are made by both team members, one will be arbitrarily chosen for grading.

9.1 Non-Standard Program Files

By default everyone is expected to use the abstract syntax tree node classes found in **AST.jar**: `ProgNode`, etc. If you were explicitly told that you could violate that expectation and design your own intermediate form, you must add information about your design in the **instructions.txt** file you submit. In addition,

You are required to provide a **Decompiler.java** *program that will display the program stored in your program files.* Again, these are the files that the developer component saves and the interpreter component reads. The operation of the decompiler should be the same as the one provided in the project web site.

9.2 How to Submit

The project is to be submitted on the mycourses system using a *Dropbox*.

Here is a checklist.

- Make sure you have cloned or pulled your latest code to to the computer on which you'll be running **mycourses**.
- Make sure you have created **instructions.txt** that explains how to interact with the two parts of the program, especially the developer side. Make sure the format is plain text rather than MS Word, HTML, PDF, or something else. This file should be added to your project and to your git repository.
- Don't forget to create **log.txt**, the Git log file.
- All of your resource files, e.g., images, must be stored within the project's directories, and your code references to them must use some kind of relative path rather than something that starts with a Windows drive letter or a UNIX forward slash (root).
- Before uploading your code the very last thing you should do is make sure your code can be built without errors or warnings and that you can successfully run the entire system! Dropboxes don't check any of this, and if we can't build or run your code, you will lose points even if you submit a corrected set of files after the due date.

Create a zip file *of your entire project*. This can be done in a shell with the command

```
zip -r augusta.zip *
```

executed from the top-level directory of the project. If using your operating system's file manager tool, selecting all of the files and choosing the "compress" or "archive" action from the menu should have the same result, *but make sure it is in the zip format*. RAR, BZIP, GZIP, TAR, JAR, etc., will not be acceptable, even though mycourses will not stop you from submitting such a file.

The name of the dropbox is "Project 2: Augusta".

10 Document Revision Notes

References to Java class `List` were removed in favor of the non-design-specific "sequence".

Noted that the provided **AST.jar** file also includes enum types and the exception class that students are supposed to use.

Added limitation that the repeat instruction count can be limited to single-digit positive numbers.

Command line specification of input files for the interpreter files are now required.

Single-stepping in the interpreter is no longer required.

Modified the Tips sections.

Increased detail in the Grading section.

Finished the Submission section.

Changed the submission instructions to use a dropbox rather than 'try'.

Listed submission requirements for teams not using **AST.jar**.

Described grading rules when more than one submission is made by a team.

Grade weights added up to 110%, so they were fixed.