For our compiler we chose to use Java. Thus we made heavy use of object oriented design concepts to effectively divide the compiler into individual and isolated modules. The compiler was split into 4 main components. The scanner, parser, weeder, and AST builder. Each with their own packages. The `Main.java` file took the input file, piped it through the scanner and then took the scanner result (a list of tokens) and piped it through the parser. It then took the result of the parser (a parse tree) and piped it through the weeder. The `Main.java` file then took the output of the weeder and piped it through the AST builder. Below is an explanation on the challenges we faced in each module and our final design for each of these modules.

Error Handling

The marking requirement for this assignment is that our program should return a Unix code of 0 if the input program is valid joos and a code of 42 if the input program is not valid joos. Thus, in order to allow for easy error handling for each of our 4 modules we wrapped all calls to these modules in Main.java with a try catch. This would catch all exceptions thrown by any of the 4 modules and would return 42 if an exception was caught else it would return 0. A separate exceptions package was used to store all custom exceptions (such as an invalid syntax exception) that could be thrown by any of the modules.

Common Data Representation Classes

In order to allow our 5 modules to operate on each others inputs we created a set of common data types that represent Tokens, parse trees, and ASTs. The `TokenType.java` class is an enum that enumerates all possible terminal and non terminal tokens for the scope of this compiler. The `Token.java` class in an interface that is implemented by both terminal and non terminal token data classes. The `TerminalToken.java` class is used to represent a terminal token (ie a token that cannot be expanded into another set of tokens using the grammar). The `NonTerminalToken.java` class is used to represent non terminal tokens. For both `TerminalToken` and `NonTerminalToken` classes we set their constructors to private. In order for a developer to create a new token they must call the getToken static method located within both `Terminal` and `NonTerminal` token classes. This method returns a new copy of the given type of `TerminalToken` or `NonTerminalToken`. If the given type is invalid (ie. it is not a terminal token or it is not a non-terminal token) then `null` is returned. This pattern allows us to specify all valid tokens in one place (in the `generateTokens` function of both `TerminalToken` and `NonTerminalToken`). For `TerminalToken`s this also allows us to specify the symbols that make up those tokens in a centralized place (single source of truth). As for the `ASTTreeNode.java` and `ParseTreeNode.java` they each represent a node within their respective trees. They simply contain references to their children.

Scanner Design: (Habeeb)

The function of a scanner is to perform the lexing step of the compiler. That is to perform the lexical analysis of a set of symbols that constitute the program we are trying to compile. A scanner should take in a set of symbols (in this case ASCII symbols) and should output a set of terminal tokens if the program forms only valid tokens else an exception should be thrown. The traditional means of making a scanner involves taking a set of regular expressions (one for each token) and converting them into a set of NFAs. We would then convert these NFAs into a single DFA. We would then run this DFA using maximal munch to eventually scan in a set of tokens from our input. It should be noted that this method is incredibly labor intensive. Converting a set of regular expressions into a DFA without the use of a program like lex and doing so by hand would take far too much time. There would be too much chance for error if it is done by hand and if tweaks had to be made the whole process would have to be repeated thus taking even more time.

This was a challenge that needed to be solved in an effective manner. It can be noted that a DFA is essentially the same thing as a set of NFAs running in parallel where we feed our input character by character into every NFA. We then check the state of every NFA to see if it is in the accepting state and keep track of the last NFA that was in an accepting state as well as the symbol at which the NFA entered into the accepting state. If an NFA does not have any transition for a given char or it has entered the accepting state we no longer feed input into that NFA. Once we have run out of NFAs to feed input to we can take a look at the last NFA that entered the accepting state and add its token to the set of tokens we will return from the scanner. We have one NFA for every token. For all keyword tokens we can use just one NFA and for the remaining types (such as literals) we can use special NFAs. Our scanner used this strategy.

While this strategy is considerably less efficient than using a pre calculated DFA it is much simpler to implement and modify.

Parser Design: (Steven)

The parser takes in a list of tokens and converts it into a parse tree. It does this by building a state machine that represents the grammar of Java. All of the necessary productions for proper Java code are encoded in `JavaGrammar.java`. The state machine is generated from this. The state machine is based on a LALR(1) state machine.

Each state in the machine consists of a set of *defining productions* and a set of *all productions*, as well as a list of integers that represents the position of the state within the right hand side of each production. To generate the state machine, the code first creates an initial state, then looks at all of the tokens that could be valid at the very start. For each such token, it creates a state, where the defining productions are all of the productions that the parser could be in based on that first input token. It also generates the list of all productions by recursively checking each token on the right hand side of the defining productions and seeing if any of its productions are the same as this state. The states have links for each valid token that they can take, each link being either a pointer to another state or a reduction. Reductions happen after a state reaches the end of the right hand side of one of its productions.

Once the state machine is generated, the parser goes through its list of tokens one by one and applies them to the machine. If its current state has a link to another state when it is given the token, the parser just changes to that state. If it reaches a reduction, states are popped from the parsers state stack based on the number of tokens in the reductions right hand side and passes the reductions left hand side as the next token. If it ever reaches a point where the next token does not have an associated link in the current state, it throws an error. To generate the parse tree, the parser continuously creates new tree nodes for each token, then adds the proper amount of nodes as children of a new node every time it reaches a reduction. The biggest challenge in creating this parser was dealing with ambiguous reductions. The states are not created with the history of where it is connected in mind, so reductions must be based on a productions left hand side token, rather than the production itself. Making it based on productions ends up creating a state machine with thousands of states. An example of how this could create an issue:

$$S \to a\ A\ d$$
$$A \to b$$
$$S \to a\ B\ c$$
$$D \to B\ d$$
$$B \to b$$

Input tokens: $a\ b\ d$

In this case, the state machine will have a reduction for $B \to b$ when the next character is either d or c. This will cause an issue when it is deciding whether to reduce $A \to b$ or $B \to b$, since both could be followed by d. Knowing what states the parser was in previously will eliminate this issue, of course, since if we came from the state that has both $S \to a \bullet A\ d$ and $S \to a \bullet B\ c$, we know that reducing $B \to b$ could only be followed by c, so if we get the token d then we must reduce using $A \to a$. Luckily this particular issue only comes up in one place in the Java specification, so simply programming a specific case is enough to fix it.

Weeding Design (Yi Fei):

The weeder works by recursively calling each node of the parse tree to visited it. At every node using a switch node it checks the node type if it is one of the nodes types that need to be checked for specific rules it will run the case block and then recursively call all its children. All the rules are implemented this way in the case block statements.

The `CLASS_DECLARATION` node checks the class name to verify that the filename is the same as the class name plus ".java". It also checks all the modifiers to make sure a class cannot be both abstract and final and also that a class has an access modifier

The `CLASS_BODY_DECLARATIONS` node checks that the class has explicit constructor by looking at its children and iterate down to find `CONSTRUCTOR_DECLARATION` node

The `METHOD_DECLARATION` node checks all the modifiers so that A static method cannot be final, An abstract method cannot be static or final,Methods must have an access modifier and also looks at the body nodes to make sure A method should have a body unless it is neither abstract nor native.

The `ABSTRACT_METHOD_DECLARATION` node check the modifies to make sure An interface method cannot be static, final, or native.

The `FIELD_DECLARATION` node checks all the modifiers to make sure fields must have an access modifier.

The `INTEGER_LITERAL` node converts the string into an integer and make sure that the integer is within limit of the java integer type.

AST Builder Design (Yi Fei):

The `ASTBuilder` works like the weeder: it accepts the root of the parse tree and recursively calls itself to visit it and change it to have a simpler structure. There are 2 kind of changes for the AST tree. The first is lists: because the parse tree wants to be unambiguous, a list of the same element will actually be a tree where the right node is one of the elements and the left node is the same type as the parent of the element. In the AST, the list of elements will be on the same level as all children of a single parent, instead of a very left heavy tree. This applies to many node types like modifiers, parameter lists, block statements and many more. The second kind of change regards specifically the expression, because we want to encode order of operations in the parse tree structure. Expression has many order like conditions: expression, relational expression, additive expression, multiplicative expression . . . this often creates unnecessary levels since when an expression derives only an integer literal it still needs to go through every level. To simplify, when the expression or any other sub node only has one child, replace the current node by its child so that in the AST an expression can be the direct parent of a integer literal.