1 Introduction

Our design contains 5 phases:

1. Tokenize the source code
2. Generate parse tree in the parser
3. Apply language restrictions in Weeder
4. Construct the AST for the parse tree
5. Testing

We will be discussing our detailed implementations in the following sections.

2 Scanning

Our Scanner Object is placed in the file '/src/main/scala/A1/Scanner.scala'. It takes an input of a seqeunces of strings and it will output a series of tokens. The Scanner Object contains 4 main components.

- DFA initializations.
- Apply Maximal Munch Algorithm onto the input string.
- Modify tokens.
- Output the sequence of tokens.

These components were computed in order, and their functionalities will be discussed in details below.

2.1 DFA Initializations

We've constructed 11 DFAs for **Comments**,**Identifiers**,**Keywords**,**Operators**,**Seperators**,**Boolean Literals**,**Null Literals**,**String Literals**,**Char Literals**, **Integer Literals** and **Whitespace Characters**. The specific accepting strings and production rule definitions are defined base on the *The Java Language Specification, Second Edition*.
The basic DFA structure is defined below:

```
case class DFA{
    name: String,
    alphabet: Set[Char] = Set.empty,
    states: Set[State] = Set.empty,
    start: State = "",
    accepting: State => Boolean,
    transition: PartialFunction[(State, Char), State] = PartialFunction.empty,
    priority: Int = 0
}
```

The priority integer is been used in cases where two DFAs both accept the same string, then the DFA with a higher priority will be used to generate the token kind. In our DFAs, only the keyword DFA is defined with a priority of 5, all other DFAs have priorities of 0. This is because if the input string is a keyword, it should only be a keyword token. No other DFAs have conflict with each others.

2.2 Maximal Munch Algorithm

Our **Maximal Munch Algorithm** is defined in **Scanner.maximalMunch**. It will generate the tokens base on 4 stages:

1. Run the input on DFAs through a loop, and record the maximal length input that's been accepted and its accepting state.
2. Compare the maximal input length between all DFAs and pick the DFA with the maximal length to tokenize. If a tie occurs, pick the DFA with a higher priority number. If the maximal length is 0, throw an error.
3. Partition the input string into two parts. The first part is the prefix of the input that's been accepted by a DFA from stage 2. The rest of the input will be used to run the **Maximal Munch Algorithm** again if not empty.
4. Terminate and output the token sequence if the whole input string has been processed and tokenized.

One issue we've faced concerns about the occurances of whitespace characters between valid tokens. Our solution is to define a whitespace DFA to handle whitespace characters. It is correct to do so because no other DFAs contain transitions from their start state to other states through a whitespace character transition.

2.3 Token Modifications

Some modifications are employed onto the seqeunces of output tokens from the **maximalMunch** function. These modifications serve for 2 purposes:

1. Remove **Comment** and **Whitespace** tokens.
2. Generalize and unify some tokens.

Our first purpose is necessary because **Comment** and **Whitespace** tokens does not contain any meanings in our grammar. No parsing rules require any use of **Comment** nor **Whitespace** tokens.

Our second purpose focus more on some special tokens:
For example: Our DFA for Integer Literals has two accepting states **ZERO** and **OTHER_INTEGERS**, these two states are both integer literals. When processing the list of tokens, we modify these token kinds to **INTEGER_LITERAL**. This provides ease in designing the LALR(1) Grammar for the language.

2.4 Output

In addition to the modified token sequence. We've appended a new token **BOF** to the front the front of the sequence, and a new token **EOF** to the end of the sequence.
BOF is used to indicate beginning of a series of tokens to be parsed.
EOF acts as a fresh terminal to avoid crashing the program when parsing.

3 Parsing

Our Parser Object is placed in the file '/src/main/scala/A1/parser.scala', it takes a sequences of tokens as input and it will output the root of the parse tree for the tokens. It contains 4 components:

1. CFG construction.
2. CFG to LALR1.
3. Read and store LALR1 parsing table.
4. Parse the tokens.

3.1 Tree

We've defined a tree class that will be used to store the parse tree and later on the AST.

```
class Tree{
   val lhs: Token
   val children: Seq[Tree]
}
Each node of the tree stores the lhs of a production rule, which is a token.
```

3.2 CFG Construction

Our CFGs are in the file '/src/main/scala/input.cfg'. These grammars has been extracted from the *The Java Language Specfication, Java SE 8 Edition* with slight modifications.
Whenever an 'opt' rule is encountered:
For example: For $A \rightarrow B_{opt}C$ where A, B, C are variables. We build two production rules $A \rightarrow BC$ and $A \rightarrow C$.
Some rules are simplified:
If we have $A \rightarrow B$ and $B \rightarrow CD$ where A, B, C, D are variables. We build the production rule $A \rightarrow CD$.
Also, we've removed some production rules defined in the textbook which is not part of JOOS1W.

3.3 CFG to LALR1

We use the provided code from the course website to perform the conversion. That code is located at '/src/main/java/jlalr1.java'. The output LALR1 grammar and the parse table is located in the file '/src/-main/scala/output.lr1'.

3.4 Read LALR1 Grammar

We've created an Object called Grammar, which is located at the file '/src/main/scala/A1/Grammars.scala'. It will read our LALR1 grammar file and output:

1. **ReduceRuleMap**
2. **ProductionRules**

ReduceRuleMap have the map structure of (int, string) $\Longrightarrow$ (string, int). This is used to store the reduce and shift rules.
For the rule **0 BOF Shift 508**, our map entry will store (0, "BOF") as its key and ("Shift", 508) as its value.

ProductionRules have the list structure of **Production**, which is defined with:

```
lhs: String
rhs: Seq[String]
```

For the production rule **S BOF CompilationUnit EOF**. The lhs of **Production** will store **S** and rhs will store the sequence of string **[BOF,CompilationUnit,EOF]**

3.5 Parse the Tokens

Our function **ParseJoos1W** located in the file '/src/main/scala/A1/Parser.scala' will take in the sequences of tokens and output the root to the parse tree. Our parsing function is based on the pseudocode:

```
1: procedure ParseLALR1(List of Tokens)
2:     for each token in Tokens do
3:         while Reduce(stack, a) = {A → γ} do
4:             pop γ off stack (pop |γ| times)
5:             push A on stack
6:     if Reject(stack + α) then
7:         Error
8:     Push α
```

We've made a modification to the above pseudocode. After poping γ off from the stack, we include them as children of A. And at the end we construct the root node S which completes our construction of a parsing tree.

4 Weeding

Our weeding object is defined in 'src/main/scala/A1/Weeding.scala'. It takes in a parse tree as its input and check for any restrictions of the Joos1W language where the parser isn't able to detect. We've included 15 different weeders in our weeder object:

AbstractClassBodyWeeder **AbstractFinalClassWeeder** **MethodModifierWeeder**

InterfaceNoFieldWeeder **InterfaceMethodModifierWeeder** **IntegerRangeWeeder**

ExplicitConstructorWeeder **NoFieldFinalWeeder** **NoMultiDimWeeder**

NoMultipleDeclarationsInForInitWeeder **NoMultipleForUpdatesWeeder**

NoExplicitSuperThisWeeder **CastWeeder** **OneTypePerFileWeeder**

PackagePrivateWeeder

We've defined a helper function **collect(tree: Tree, lhsKind: String): Seq[Tree]** to return a sequence of nodes which contain the **lhsKind** token kind.

4.1 Challenges

It is possible to encounter cases where some rules cannot be written in LALR(1). One lookahead is not enough. We've encountered a problem with casting.
Suppose that we are given the following statements:

```
super((test))
super((test)4)
```

The first line treats test as an parenthesized expression, however in the second line, test should be treated as a reference type for casting. It is not possible to identify the type with 1 lookup. The solution is to treate both cases as an expression type. And leave the testings to the weeding stage.

4.2 Not JOOS1W Constructions

We've removed non JOOS1W grammars from our Context Free Grammar because It will reduce the amount of work that need to be placed in the weeder.

5 AST Building

Our AST Building object is defined in 'src/main/scala/A1/ASTBuilding.scala'. It will take in the root of a tree as its input and output an AST.
Our pseudocode for this method construction is defined as following:

```
1:  procedure ASTBuild(Tree root)
2:      if The root has no child then
3:          return root
4:      else if The root has one child then
5:          return ASTBuild(root's child)
6:      else
7:          new_children = []
8:          for each child in root.children do
9:              new_children.append(ASTBuild(child)
10:         root.children = new_children
11:         Return root
```

6 Testing

We've implemented 3 stages of testings through the **scalatest.FunSuite** package. Our focus is on Scanner, Parser and Weeder. We've included a DFA test file, a Weeder test file and a full test file. In the full test file, we've tested the Scanner, Parser and Weeder components.

6.1 Scanner Tests

To ensure our DFA is correct, we've tested each DFA component individually. The tests are included in 'src/main/scala/test/scala/DFATests'.

6.2 Weeder Tests

We've included syntax correct code snippets in 'src/main/scala/test/scala/WeederTests'. For each Weeder, they will check for language restrictions.

6.3 Full Test

In this section, we've included 320 different Java Source Code for testing. They've been extracted from the course '/resource' folder.

The files has been classified into two different categories, 'pass' and 'reject'.

All 'pass' category source code files should be accepted with no exceptions.
All 'reject' category source code files should be rejected by either the Scanner, the Parser or the Weeder.
If they've been rejected, a message of error type will be printed and can be checked to match up with their expected failures.