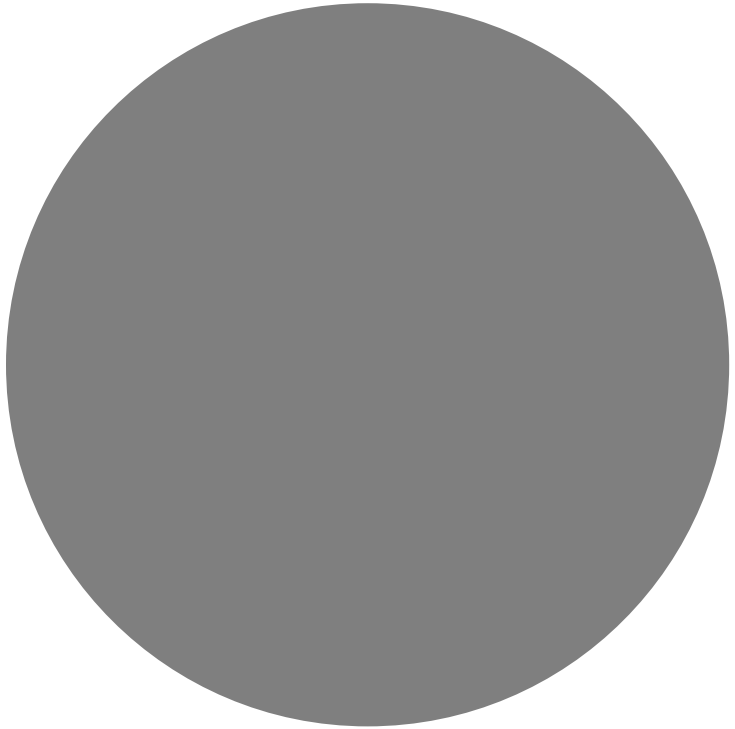
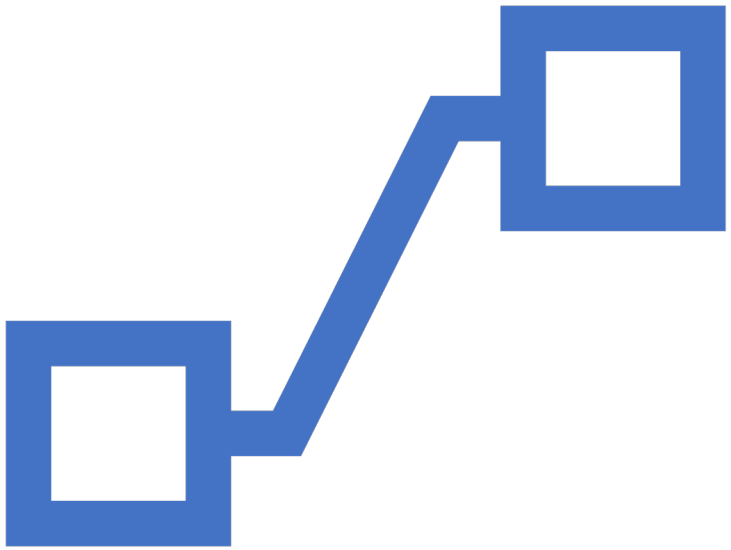
CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023





Semantic Analysis

Programming Assignment #4
Introduction

Role of These Slides

- These slides are to *supplement* the `README.md` project description in the GH repo starter code, as well as the lectures
- Consult all resources and ask questions on discord
- See these slides for a visual presentation of some of the subtleties in the assignment

Overall Task

- You'll be implementing semantic analysis into your compiler
- Involves:
  1. Symbol table construction
  2. Type checking

- The `README.md` lists 8 tasks in more detail

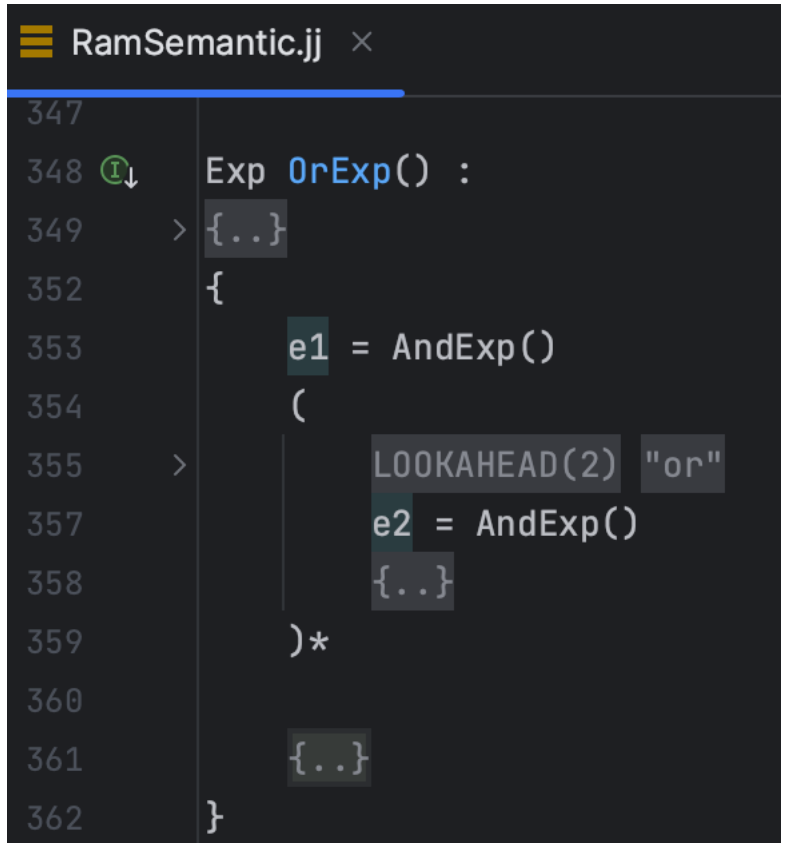
Project Structure

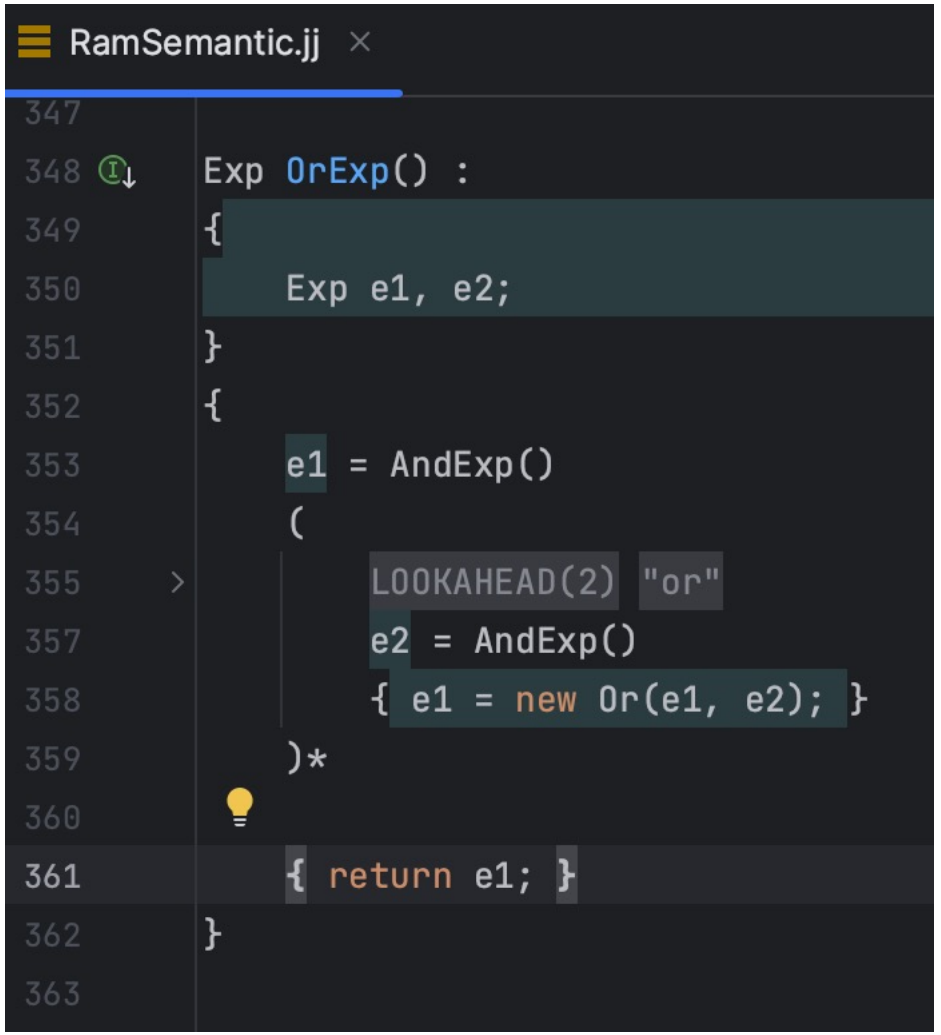
- There are some new files in this starter code, including
- A 99% complete `RamSemantic.jj` file, for the Ram23 grammar specification
  - … so if you didn't finish Programming Assignments #2 or #3, everyone is starting from the same spot for this assignment

RamSemantic.jj

- This grammar files that I've given you in the starter code already includes semantic actions, so that JavaCC will automatically build an Abstract Syntax Tree (our intermediate representation), which we'll traverse to build the symbol table and perform type checking
- I've also augmented the grammar so that precedence rules are appropriate captured (e.g. * multiplication is deeper than + addition, and && occurs before ||).
  - Result: creation of more non-terminals than what you had in Programming Assignment #3 (e.g. `OrExp()` – see next slide)

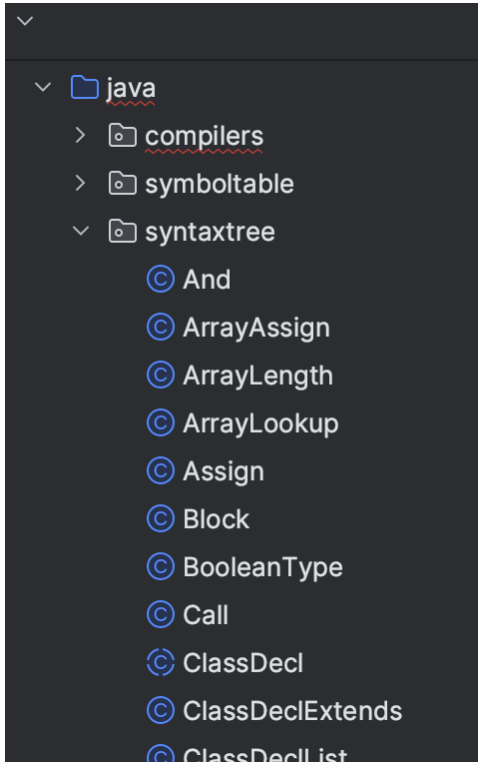
IntelliJ may automatically *hide* these semantic actions by default





Remember that semantic actions are Java code that is run in parallel when this production is used. So, the `OrExp()` *non-terminal returns an* `Exp` *object, with Java code interspersed on* Lines 348, 350, 358, 361.

- The referenced `Or` constructor on Line 358 is referring to the class `java.syntaxtree.Or`
- The `syntaxtree` classes are imported into `RamSemantic.jj` on Line 11, and will also be imported into the generated `RamParser.class` when the javacc action is run by Maven



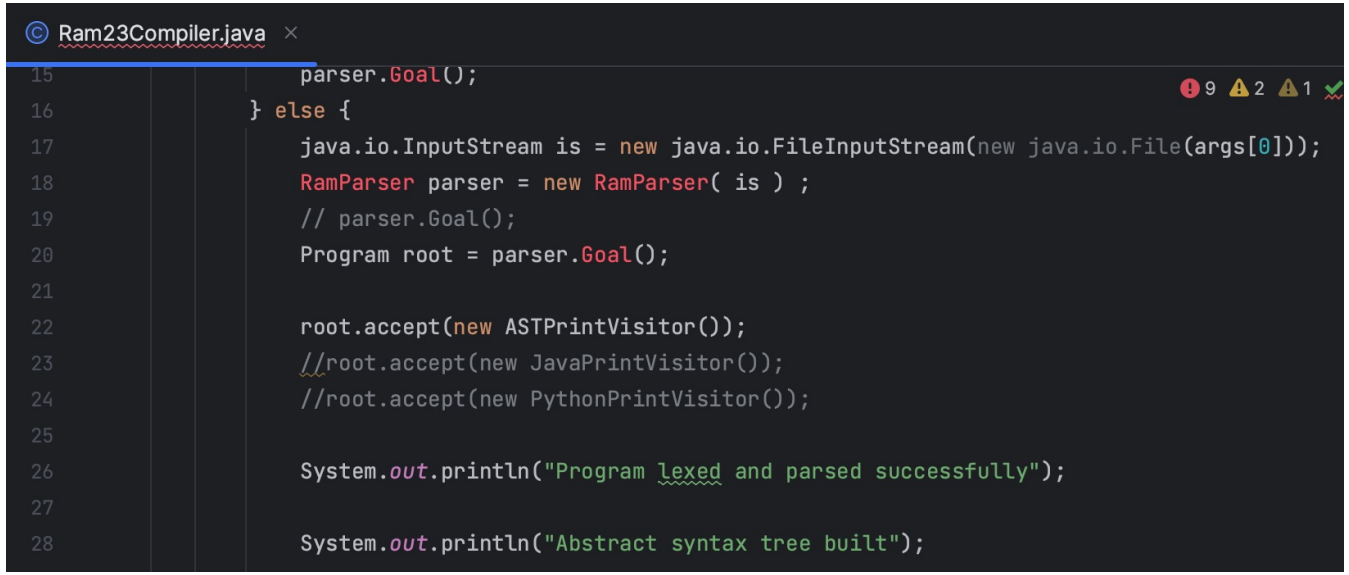
RamSemantic.jj

```
options {
    JAVA_UNICODE_ESCAPE = true;
    DEBUG_PARSER = false;
    STATIC = false;
}

PARSER_BEGIN(RamParser)
    package compilers;
    import java.io.FileNotFoundException;
    import visitor.*;
    import syntaxtree.*;
```

RamSemantic.jj

```
Exp OrExp() :
{
    Exp e1, e2;
}
{
    e1 = AndExp()
    (
        LOOKAHEAD(2) "or"
        e2 = AndExp()
        { e1 = new Or(e1, e2); }
    )*

    { return e1; }
```

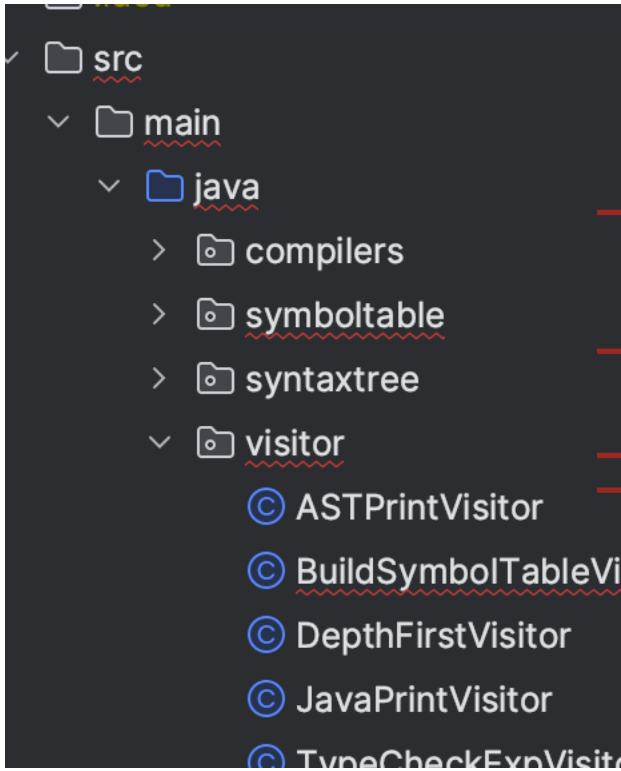
An initial experiment



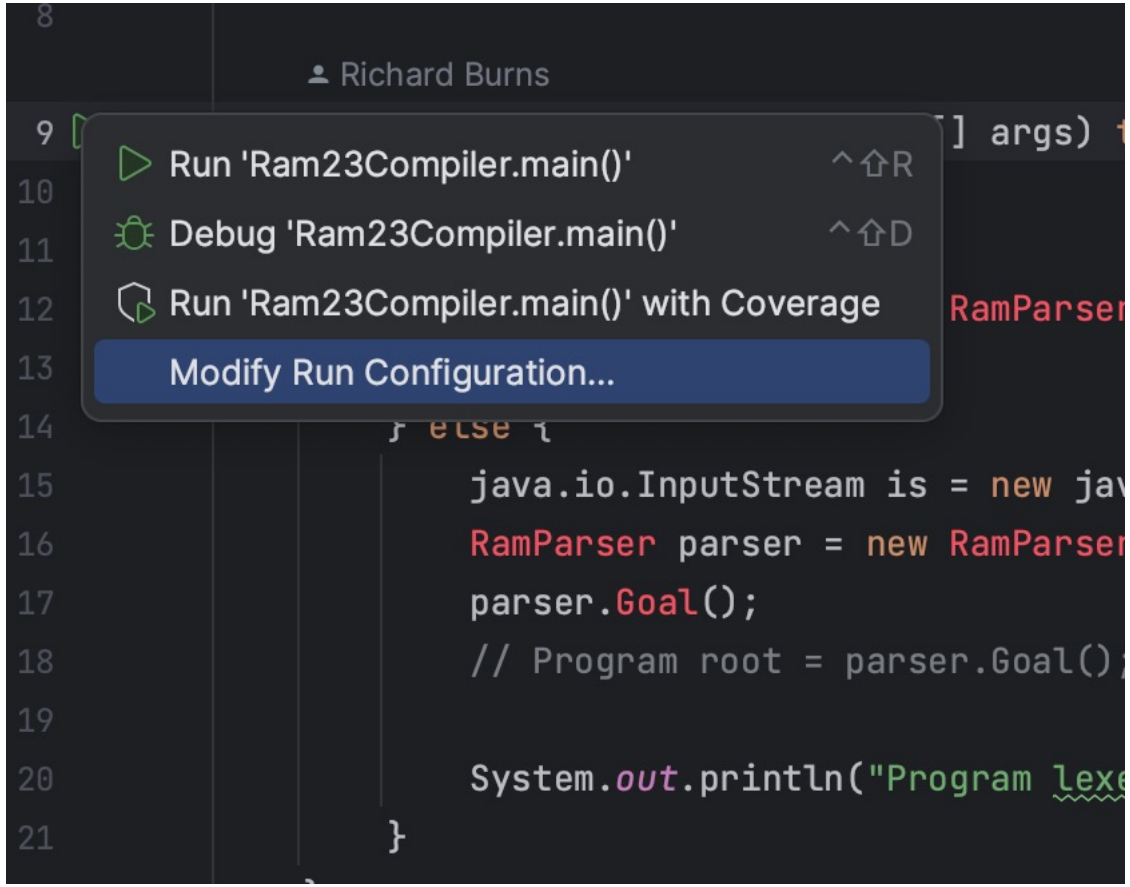
- Perform a `mvn clean`, and `javacc:jaccc` to generate the `RamParser.class`
- See `src/main/java/compilers/Ram23Compiler.java`
- Notice that the `Ram23Compiler` is now extended on Lines 22-46 to do additional tasks once the *Parser* finishes
  - The parser returns the constructed AST for a source program on Line 20, which we'll traverse (using the *Visitor* design pattern) to perform other tasks

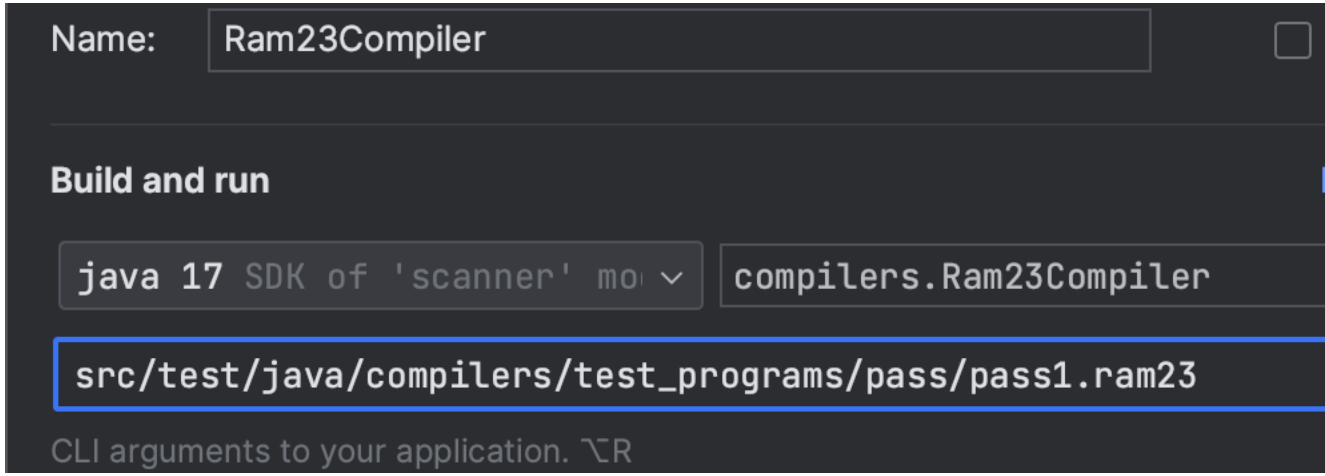
Feel free to Comment Out Additional Lines

- I suggest commenting out the symbol table and type checking calls initially (Lines 31-46) and just experiment with the Visitors
- Explore the following classes:
  - `java.visitor.ASTPrintVisitor` (called on Line 22)
  - `java.visitor.JavaPrintVisitor` (called on Line 23)
- These visitors will perform the tasks of "pretty printing" the generated AST and exporting the AST as a legal `.java` program
  - So, Line 23 is the big last step in a full Ram23→Java conversion
  - **Really cool!**



Set the *run configuration* to use `Ram23Compiler` on a single file





Output Example

Input arg: `src/test/java/compilers/test_programs/pass/HelloWorld.ram23`

Output of `ASTPrintVisitor`

```
Program(
MainClass(Identifier(HelloWorld), Identifier(a), Println(IntegerLiteral(1)))
ClassDeclList(
))
```

Output of `JavaPrintVisitor`

```
class HelloWorld {
  public static void main (String [] a) {
    System.out.println(1)
  }
}
```

```
Program lexed and parsed successfully
Abstract syntax tree built

Process finished with exit code 0
```

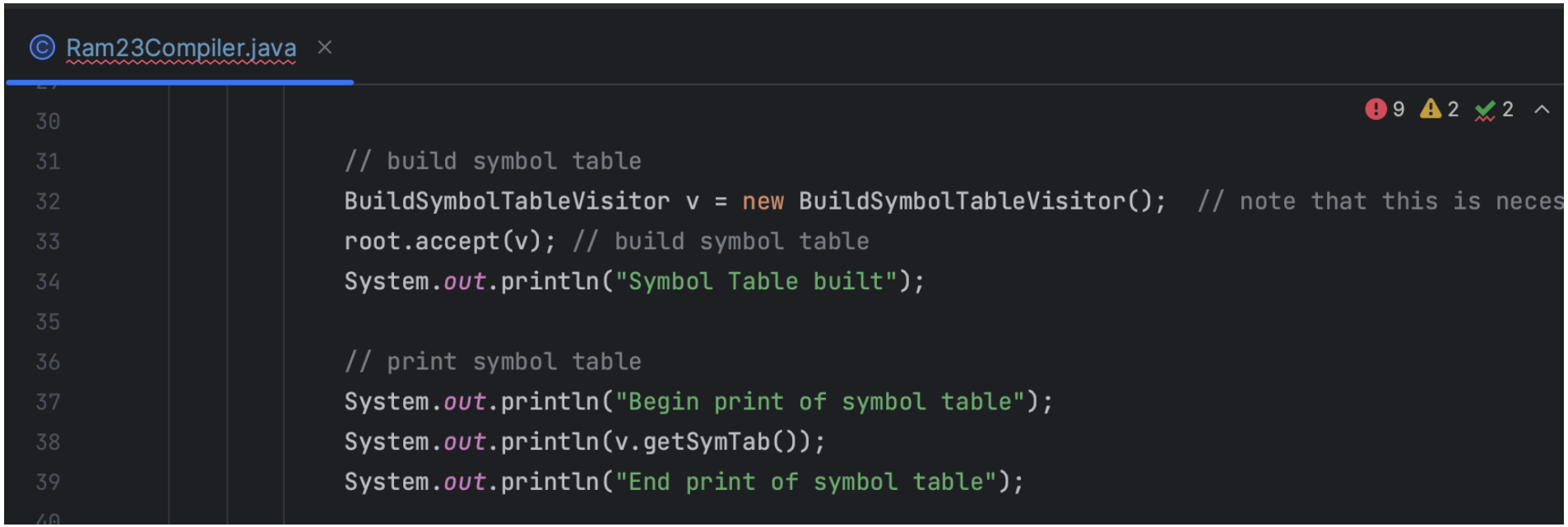
(it looks like my JavaPrintVisitor has a bug and is missing the emit of a `;` at the end of the Println AST!)

Before proceeding, make sure you are on your way to getting comfortable with *how* and *why* these Visitors work and do their job.

Next Steps

- At this point, you are ready do start looking at and thinking about Tasks #1-4 in the `README.md`, in which you'll:
  - [take a look at these; I've already written these classes and they are 99% complete]
    - `java.visitor.BuildSymbolTable`
    - `java.symboltable.Table`
    - `java.symboltable.RamClass`
  - [create and write these yourself while consulting Task #4 of README.md]
    - `java.symboltable.RamMethod`
    - `java.symboltable.RamVariable`

Uncomment symbol table lines from `Ram23Compiler`



- Lines 32-33 construct the symbol table (Task #4)
- Line 38 “pretty prints” the created symbol table (Task #5)

Task #6: Type Checking

- See `README.md` for a description of the type checking phase of this assignment.
- Appropriate implement a new Visitor (`java.visitor.TypeCheckVisitor`) which takes the symbol table as an argument in its constructor (Line 43)
  - Why? -- The type checker needs to know which variables are declared and what their types are.

```
42          // perform type checking
43          root.accept(new TypeCheckVisitor(v.getSymTab()));
44          if (v.getSymTab().anyErrors())
45              throw new ParseException ("SA error detected");
46          System.out.println("Semantic Analysis: Type Checking complete");
47
```