

Compiler Project PA2 – Abstract Syntax Trees & Operator Precedence

Due: 7/9/24 11:59:59pm

This is the second programming assignment in your journey to making a compiler. The goal for this milestone is to create Abstract Syntax Trees (ASTs) for syntactically valid miniJava programs. This checkpoint will build on your implementation in PA1. There will be some small grammar changes that you will need to derive.

Abstract Syntax Trees effectively store a valid program's structure such that we can iterate through the program's syntax. We are beyond simply checking syntax validity but now need to store the source code meaningfully using ASTs.

1. Operator Precedence

Ignoring conditional branching, when we see multiple statements, we can safely assume that earlier statements execute before later statements. However, within a statement, that is not necessarily the case. Consider the following expression:

$$1 + 3 * 4 / 2$$

Earlier operations in this expression are NOT executed first. Instead, we have precedence rules. Let's rewrite the above expression showing precedence.

$$1 + ((3 * 4) / 2)$$

Although multiply and divide have the same precedence in miniJava, the multiply expression is processed first because we process left-to-right. The table below lists miniJava operators in order of lowest precedence to highest.

Class	Operators
Disjunction	
Conjunction	&&
Equality	==, !=
Relational	<=, <, >, >=
Additive	+, -
Multiplicative	*, /
Unary	-, !

The highest precedence expression would be what is left over (LiteralExpr, NewObjectExpr, NewArrayExpr, IxExpr, CallExpr, RefExpr, and parenthesized Expression, see page 3 for definitions). To effectively process operators in precedence order using recursive descent, we must construct a stratified grammar (see Lecture 6).

2. Abstract Syntax Trees

The actual Abstract Syntax Tree objects are mundane to implement. As such, we will provide all the AST objects you need for PA2 on the course website. Your first step in your Compiler should be to import the AST java files. First, create a package called `miniJava.AbstractSyntaxTrees` and then import all of the AST java files there.

If you are not tracking line numbers for each Token in your Lexer, create an empty `SourcePosition` class inside `miniJava.SyntacticAnalyzer`. Tracking line numbers is highly recommended, extra credit in PA5, but nonetheless optional. If you choose not to track them, whenever a `SourcePosition` is necessary for the creation of an AST, simply pass null for the `SourcePosition`. Additionally, create a method inside your `Token` object called `public SourcePosition getTokenPosition()`. This method should return `null`.

If you are already tracking line numbers or wish to start, you must package those numbers into the `SourcePosition` class inside `miniJava.SyntacticAnalyzer`. The `SourcePosition` object can be initialized however you wish (for example, with a line and column number), but must implement the `toString()` override method where it will return a `String` indicating the source code position. Next, your `Token` class should be initialized with a `SourcePosition` object and stored. Ensure a public method called `getTokenPosition()` that returns the stored `SourcePosition` exists. In your `Scanner` implementation, you will likely only need to modify `makeToken` and `nextChar` where you detect line breaks in `nextChar`, and create a `SourcePosition` for the token generated in `makeToken`. If your implementation differs from the PA1 starter code, ensure that `Tokens` are created with `SourcePosition`.

When troubleshooting your PA2 implementation, modify `ASTDisplay.java` and change `showPosition` to `true`. This will output your `SourcePosition` in the AST visual output. However, when submitting, ensure `showPosition` is set to **false**, otherwise the autograder will not be able to validate your AST implementations.

On the next page, we show the grammar rules once again, but also show which AST implementation that rule corresponds to.

Program	::= ClassDeclaration * eot	Package
ClassDeclaration	::= class id { (FieldDeclaration MethodDeclaration)*}	ClassDecl
FieldDeclaration	::= Visibility Access Type id ;	FieldDecl
MethodDeclaration	::= Visibility Access (Type void) id (ParameterList ?) { Statement * }	MethodDecl
Visibility	::= (public private)?	n/a
Access	::= (static)?	n/a
Type	::= int boolean id (int id)[]	TypeDenoter
ParameterList	::= Type id (, Type id)*	ParameterDeclList
ArgumentList	::= Expression (, Expression)*	ExprList
Reference	::= id this Reference . id	IdRef ThisRef QualRef
Statement	::= { Statement * } Type id = Expression ; Reference = Expression ; Reference [Expression] = Expression ; Reference (ArgumentList ?) ; return (Expression)? ; if (Expression) Statement (else Statement)? while (Expression) Statement	BlockStmt VarDeclStmt AssignStmt IxAssignStmt CallStmt ReturnStmt IfStmt WhileStmt
Expression	::= Reference Reference [Expression] Reference (ArgumentList ?) unop Expression Expression binop Expression (Expression) num true false new id() new (int id) [Expression]	RefExpr IxExpr CallExpr UnaryExpr BinaryExpr Expression LiteralExpr (IntLiteral) LiteralExpr (BooleanLiteral) NewObjectExpr NewArrayExpr

Terminals are in bold.

3. AST Specifics

This assignment will require you to inspect the files inside the provided `AbstractSyntaxTrees` package. Consider `WhileStmt ::= while (Expression) Statement`.

```
public class WhileStmt extends Statement
{
    public WhileStmt(Expression e, Statement s, SourcePosition posn){
        super(posn);
        cond = e;
        body = s;
    }

    public <A,R> R visit(Visitor<A,R> v, A o) {
        return v.visitWhileStmt(this, o);
    }

    public Expression cond;
    public Statement body;
}
```

Because a `WhileStmt` takes a conditional expression and a statement (which can be a `BlockStmt` if it is multiple lines), it is initialized with exactly those ASTs, an `Expression` and a `Statement`.

There are some trickier details where a `MethodDecl` takes a `FieldDecl` (for the `Visibility`, `Access`, `Type`, `id`), and tacks on a `ParameterDeclList` and a `StatementList`. Similarly, `LiteralExpr` has to be initialized with either a `BooleanLiteral` or `IntLiteral`. If a corresponding “else” does not exist for `IfStmt`, then the `Statement` for the “else” parameter in `IfStmt` should be null. For this reason, before creating the AST in question, you may want to review the corresponding AST implementation.

4. Programming Assignment

Your `Parser`’s `parse` method should return an AST object (specifically `Package`). Similarly, your `parse` methods, such as `parseType`, should return the associated AST object, such as `TypeDenoter`. `Parse` methods with multiple ASTs associated with them should return an abstract version of that AST. For example, `parseStatement` should return `Statement`, but the method itself may create a `BlockStmt`, `ReturnStmt`, `WhileStmt`, etc.

The main method inside `Compiler.java` should no longer output “Success” when there are no syntax errors. Instead, initialize an `ASTDisplay` object and call `showTree` on the AST returned by your `parse` method. Do not output anything else when there are no errors. When there is a syntax error, ensure “Error” is output on the first line by itself as usual, then output any error messages.

```

class PA2Sample {
    public boolean c;
    public static void main(String[] args) {
        if (true)
            this.b[3] = 1 + 2 * x;
    }
}

```

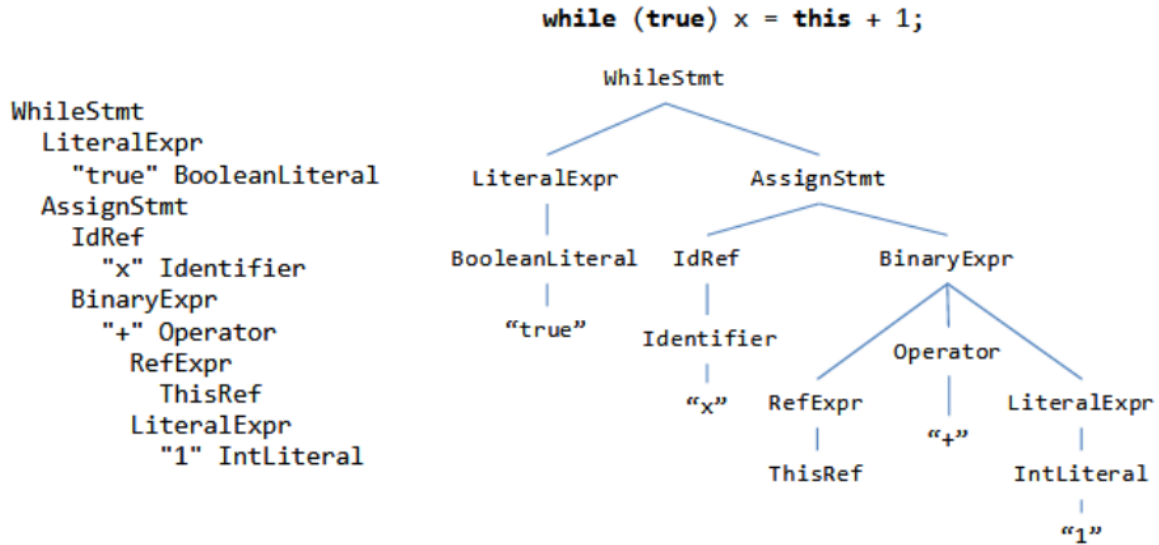
The above code is syntactically valid (even if `this.b` is not defined) and results in:

```

===== AST Display =====
Package
ClassDeclList [1]
. ClassDecl
. "PA2sample" classname
. FieldDeclList [1]
. . (public) FieldDecl
. . BOOLEAN BaseType
. . "c" fieldname
. MethodDeclList [1]
. . (public static) MethodDecl
. . VOID BaseType
. . "main" methodname
. . ParameterDeclList [1]
. . . ParameterDecl
. . . ArrayType
. . . ClassType
. . . "String" Identifier
. . . "args" parametername
. . StmtList [1]
. . . IfStmt
. . . LiteralExpr
. . . "true" BooleanLiteral
. . . IxAssignStmt
. . . QualRef
. . . "b" Identifier
. . . ThisRef
. . . LiteralExpr
. . . "3" IntLiteral
. . . BinaryExpr
. . . "+" Operator
. . . LiteralExpr
. . . "1" IntLiteral
. . . BinaryExpr
. . . "*" Operator
. . . LiteralExpr
. . . "2" IntLiteral
. . . RefExpr
. . . IdRef
. . . "x" Identifier
=====

```

The above text is generated by the Visitor called `ASTDisplay` (already implemented for you). Visitors initiate a depth-first traversal of the source code passed to your compiler. Every time `ASTDisplay` visits a contained (child node) element, it requires that element output any text with more indentation:



This output determines whether your compiler created the AST correctly, including checking proper operator precedence rules.

Note: “Package” will not resolve automatically. Inside your Parser object, ensure that you import specifically `miniJava.AbstractSyntaxTrees.Package` alongside `miniJava.AbstractSyntaxTrees.*`, where the latter makes referencing ASTs easier.

When submitting to the autograder, ensure that your folder names and filenames do not have whitespace in them. Your files may have whitespace in them, but the names cannot. Ensure that “`showPosition`” is set to false inside `ASTDisplay.java`.