COMP 520: Compiler Project
PA2 – Abstract Syntax Tree Construction

Assigned: Tue Feb 8
Due: Sun Feb 27, 11:59 PM

The second milestone in the compiler project is to create an abstract syntax tree (AST) for syntactically valid miniJava programs. You will need to add code to your parser to construct the miniJava AST using a set of AST classes outlined in this document and available as a package on our course website. You will also need to make a couple of small changes to the miniJava grammar and modify the grammar to build the correct AST reflecting Java operator precedence rules.

1.   Operator precedence in expressions

The evaluation order of Java expressions is specified using parentheses and by standard operator precedence rules from arithmetic and predicate logic. The following table lists the precedence order of the miniJava operators from lowest to highest.

class	operator(s)
disjunction	`||`
conjunction	`&&`
equality	`==, !=`
relational	`<=, <, >, >=`
additive	`+, -`
multiplicative	`*, /`
unary	`-, !`

Binary operators are left associative and reflect precedence, so that $1-2+3$ means $(1-2)+3$, and $1+3*4/2$ means $1+((3*4)/2)$. Unary operators are right associative. The challenge in this part of the assignment is to construct a *stratified grammar* (Lecture 07) reflecting the precedence shown above as well as accommodating explicit precedence specified using parentheses. Your grammar can be used to build the correct AST in the course of parsing.

2. Abstract syntax tree classes

The set of classes needed to build miniJava ASTs are provided in the `AbstractSyntaxTrees` package available through the course website. Components of the AST "grammar" are organized by the class hierarchy shown on the last page (right side). Abstract classes (shown with an "A" superscript next to the class icon) represent nonterminals of the AST grammar, such as *Statement*.

The rule for *Statement* below shows the particular kinds of statements that may be created in an AST; each corresponds to a concrete AST class shown on the right. For example, a *WhileStmt* is a specific kind of *Statement*, and consists of an *Expression* (the condition controlling execution of the loop) and a *Statement* (the body of the loop).

Statement	::=		
		Statement*	BlockStmt
	|	VarDecl Expression	VarDeclStmt
	|	Reference Expression	AssignStmt
	|	Reference Expression Expression	IxAssignStmt
	|	Reference Expression*	CallStmt
	|	Expression?	ReturnStmt
	|	Expression Statement Statement?	IfStmt
	|	Expression Statement	WhileStmt

If we look inside the AST class *WhileStmt* we find the following:

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

    public Expression cond;
    public Statement body;
}
```
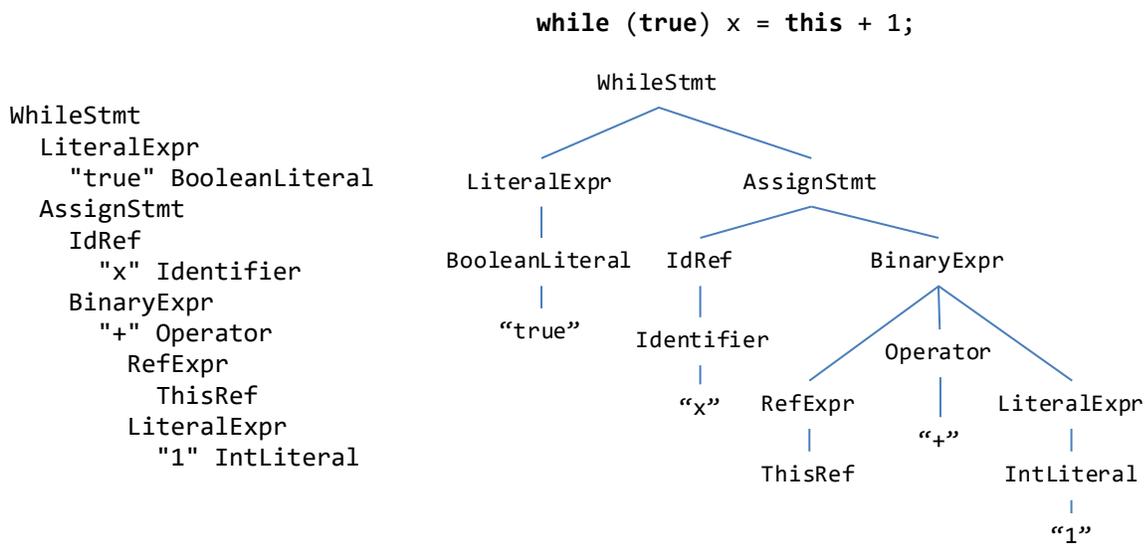
The constructor creates a `WhileStmt` node, and its two fields provide access to the AST subtrees of the node (the expression `cond` controlling the loop repetition, and the statement `body` to be executed in each repetition). Note the nomenclature, each kind of Statement has a particular name suggesting its kind (e.g. "`While`") that is joined to "`Stmt`" to show the nonterminal from which it derives.

Consult the documentation, source files, and AST constructed for the sample program to make sure you understand the contents and structure of the AST classes. Some auxiliary classes are included to provide a convenient way to create lists of nonterminals such as the `StatementList` in the `BlockStmt`. The "start symbol" of the AST grammar is `Package`. All of these are shown in the type hierarchy on the last page.

4. The AST Visitor

The `AbstractSyntaxTrees` package defines a `Visitor` interface that can be implemented to create AST traversals. Contextual analysis and code generation will be structured as AST traversals. `ASTDisplay` is an AST traversal implemented in the `AbstractSyntaxTrees` package to print a textual representation of an AST (or any AST subtree). Use this facility to inspect the ASTs you generate.

The text representation is created by a depth-first traversal of the AST. A node is displayed as its class name on a single line. The attributes and children of the node are shown in subsequent lines, indented two spaces to the right. Since the traversal is depth-first, the entire representation of the left subtree will be shown before starting the representation of the right subtree. For example, the Statement below has the AST shown on the right with text representation of the AST on the left. Note that near the leaves the text representation is compacted somewhat to improve readability.

while (**true**) x = **this** + 1;

```
WhileStmt
  LiteralExpr
    "true" BooleanLiteral
  AssignStmt
    IdRef
      "x" Identifier
    BinaryExpr
      "+" Operator
      RefExpr
        ThisRef
      LiteralExpr
        "1" IntLiteral
```


`ASTDisplay` can also list the source positions for each AST node if you enable the capability within `ASTDisplay` and provide an appropriate `toString()` method for `SourcePosition`. For these values to be meaningful, you need to correctly set the source position for every AST node in the parser. It is useful for every AST node to have an associated source position (really an interval in the source text) that can be used for error reporting in later stages. At this stage it is not required and will, by default, not be displayed in the AST. However to create an AST you will have to provide at least a null `SourcePosition` for each node.

5. Programming Assignment

For PA2 your `Compiler` mainclass should determine if the input source file constitutes a syntactically valid miniJava program as defined by PA1 and definitions above. If so, it should display the constructed AST using the `showTree` method in the `ASTDisplay` class, and terminate via `System.exit(0)`. (Note: in your submission, disable the display of source position). If the input source file is not syntactically valid miniJava, you should write a diagnostic error message and terminate via `System.exit(4)`. You may output any additional information you wish, but do not alter any aspect of the `AbstractSyntaxTrees` package for the PA2 submission. Submission testing will check that valid miniJava programs construct the correct AST. The type hierarchy for the AST classes is shown on the next page, as well as the ASTDisplay for a small sample miniJava program.

```
// simple PA2 example
class PA2sample {

    public boolean c;

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

```
======= 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
================================================
```

Example miniJava program and AST

AbstractSyntaxTrees
class hierarchy