COMP 520: Compilers
Compiler Project - Assignment 2

Assigned: Thu Jan 31, 2013 (updated Feb. 7)
Due: Tue Feb 19, 2013

The second milestone in the compiler project is to create an abstract syntax tree (AST) for any miniJava program that is syntactically valid according to our miniJava grammar. This assignment requires you to incorporate Java operator precedence rules and to build a correct AST using a set of AST classes outlined in this document and available as a package on our course website.

The current document will change, as there will be some small changes in the AST classes not reflected in the example on the last page. But the objectives for this checkpoint are fully defined and will not change.

1. miniJava syntax changes

The grammar for this assignment is the miniJava grammar from the first assignment. However, you should no longer allow `"--"` to be parsed as two subtraction operators. In full Java `"--"` is a prefix and postfix operator applied to a variable to predecrement or postdecrement the value of a variable referenced in an expression, respectively. Since we will not implement this operator in miniJava, any expression involving `"--"` should be disallowed in miniJava. Here are some examples.

Valid miniJava expressions:

```
-b    -(-b)       - -b        a-(-b)      !b    !!b
```

Invalid miniJava expressions (but valid Java expressions)

```
--b   a - --b     a---b       a--+--b
```

2. Operator precedence in expressions

In Java the evaluation order of expressions is controlled by 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, 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 reflecting the precedence shown above that also

accommodates explicit precedence using parentheses. The correct AST can be constructed in the course of parsing such a grammar.

3. 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 class in the hierarchy. For example, a WhileStmt is a specific kind of *Statement*, and consists of an *Expression* (for the condition controlling execution of the loop) and a *Statement* (for the body of the loop).

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

If we look inside the concrete class WhileStmt we find the following:

```
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`") 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. Note the classes make use of Java 1.5 features (enums, generics, and the extended `for` statement). 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`. A legal miniJava program should correspond to an AST with a `Package` root node that will contain a list of children, each of which is a `ClassDecl`.

3. The AST Visitor

The `AbstractSyntaxTrees` package includes an *interface* describing a visitor to traverse an AST and an `ASTDisplay` *implementation* of the visitor to display an AST (or any AST subtree) in text form. Use this facility to inspect the ASTs you generate. The AST display will also list the source positions for each AST node if you enable the capability in `ASTDisplay` and provide an appropriate `toString()` method for `SourcePosition`. For these values to be meaningful, you need to set the source position correctly in the parser. It is useful for every AST node to have an associated source position that can be used for error reporting in later stages, but 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 a `SourcePosition` for each node (which can be null).

An example program PA2 is shown on the last page on the left followed by its AST, as displayed using `ASTDisplay.` Note the AST classes and the example will be changed slightly in the next few days.

4. Programming Assignment

For PA2 you will make modifications in the `miniJava.SyntacticAnalyzer` package to construct a correct AST using the supplied `miniJava.AbstractSyntaxTrees` package. Your Compiler mainclass should determine if the input file constitutes a syntactically valid miniJava program as defined by PA1 and definitions above. If so, it should should display the AST constructed (using the `showTree` method supplied in the `ASTDisplay` class), and then Ssytem.`exit(0)`. (Note: as distributed, `ASTDisplay` does not display source position, but it is an option that can be enabled - do not enable it in your submission!). If the input file is syntactically invalid, you should write a diagnostic error message and terminate via `System.exit(4)`. You may output any additional information you wish from your compiler.

For valid miniJava programs the testing will check that you return exit code 0 and that the AST you display matches the expected AST, and for invalid programs it will check that you return exit code 4.

```
// simple PA2 example
class PA2 {

    public boolean c;

    public static void main(String[] args){
        int x = 3;
        if (x > 1)
            x = 1 + 2 * x;
        System.out.println(x);
    }
}
```

```
======= AST Display ========================
Package
  ClassDeclList [1]
  . ClassDecl
  .   "PA2"
  .   FieldDeclList [1]
  .   . (public) FieldDecl
  .   .   BOOLEAN BaseType
  .   .   "c"
  .   MethodDeclList [1]
  .   . (public static) MethodDecl
  .   .   VOID BaseType
  .   .   "main"
  .   .   ParameterDeclList [1]
  .   .   . ParameterDecl
  .   .   .   ArrayType
  .   .   .     ClassType
  .   .   .       "String"
  .   .   .   "args"
  .   .   StmtList [3]
  .   .   . VarDeclStmt
  .   .   .   VarDecl
  .   .   .     INT BaseType
  .   .   .     "x"
  .   .   .   LiteralExpr
  .   .   .     "3" IntLiteral
  .   .   . IfStmt
  .   .   .   BinaryExpr
  .   .   .     ">" Operator
  .   .   .       RefExpr
  .   .   .         QualifiedRef
  .   .   .           "x" Identifier
  .   .   .       LiteralExpr
  .   .   .         "1" IntLiteral
  .   .   .   AssignStmt
  .   .   .     QualifiedRef
  .   .   .       "x" Identifier
  .   .   .     BinaryExpr
  .   .   .       "+" Operator
  .   .   .         LiteralExpr
  .   .   .           "1" IntLiteral
  .   .   .         BinaryExpr
  .   .   .           "*" Operator
  .   .   .             LiteralExpr
  .   .   .               "2" IntLiteral
  .   .   .             RefExpr
  .   .   .               QualifiedRef
  .   .   .                 "x" Identifier
  .   .   . CallStmt
  .   .   .   QualifiedRef
  .   .   .     "System" Identifier
  .   .   .     ."out" Identifier
  .   .   .     ."println" Identifier
  .   .   .   ExprList [1]
  .   .   .   . RefExpr
  .   .   .   .   QualifiedRef
  .   .   .   .     "x" Identifier
==========================================
```