You are to write a Java CUP parser specification to implement a CSX parser. A grammar that
defines CSX's syntax appears below. You should examine the grammar carefully to learn the
structure of CSX constructs. In most cases, structures are very similar to those of Java and C++. Note
that at this stage you need not understand exactly what each construct does, but rather just
what each construct looks like.

The CSX grammar listed below encodes the fact that the unary ! and type cast operators
have the highest precedence. The * and / operators have the next highest precedence. The + and
− operators have the third highest precedence. The relational operators (==, !=, <, <=, >= and
>) have the fourth highest precedence. The boolean operators (&& and ||) have the lowest pre-
cedence. Thus !A+B*C==3 || D!=F is equivalent to the following fully-parenthesized expres-
sion: ((((!A)+(B*C))==3) || (D!=F)). All binary operators are left-associative, except
the relational operators which do not associate at all (i.e., A==B==C is illegal). The unary oper-
ators are (of course) right-associative. Be sure that your parser properly reflects these prece-
dence and associativity rules. Note that the increment and decrement operators (++ and --) are
not part of expressions. Rather, they are only used at the statement level.

```
program   →  class id { memberdecls }
memberdecls →  fielddecl memberdecls
              | methoddecls
fielddecls →  fielddecl fielddecls
            | λ
methoddecls →  methoddecl methoddecls
             | λ
optionalSemi →  ;
              | λ
methoddecl →  void id ( ) { fielddecls stmts } optionalSemi
             | void id ( argdecls ) { fielddecls stmts } optionalSemi
             | type id ( ) { fielddecls stmts } optionalSemi
             | type id ( argdecls ) { fielddecls stmts } optionalSemi
argdecls →  argdecl , argdecls
         | argdecl
argdecl →  type id
        | type id [ ]
fielddecl →  type id ;
            | type id = expr ;
            | type id [ intlit ] ;
            | const id = expr ;
```
stmts → stmt stmts
| λ.

stmt → if ( expr ) stmt
| if ( expr ) stmt else stmt
| while ( expr ) stmt
| id : while ( expr ) stmt
| name = expr ;
| name ++ ;
| name -- ;
| read ( readlist ) ;
| print ( printlist ) ;
| id ( ) ;
| id ( args ) ;
| return ;
| return expr ;
| break id ;
| continue id ;
| { fielddecls stmts } optionalSemi

type → int
| char
| bool

args → expr , args
| expr

readlist → name , readlist
| name

printlist → expr , printlist
| expr

expr → expr || term
| expr && term
| term

term → factor < factor
| factor > factor
| factor <= factor
| factor >= factor
| factor == factor
| factor != factor
| factor

factor → factor + pri
| factor - pri
| pri

pri → pri * unary
| pri / unary
| unary

unary → ! unary
| ( type ) unary
| unit

unit → name
| id ( )
| id ( args )
| intlit
Using Java CUP to Build a Parser

You will use Java CUP, a Java-based parser generator, to build your CSX parser. You’ll have to rewrite the CSX grammar into the format required by Java CUP. This format is defined in “CUP User’s Manual,” available in the “Useful Programming Tools” section of the class home-page. A sample CUP specification corresponding to CSX lite (a small subset of CSX) is at www.cs.wisc.edu/~fischer/cs536.s16/course/proj3/startup/java/lite.cup. The file lite.cup is also included in the Eclipse archive for project 3.

Once you’ve rewritten the CSX grammar we’ve provided and entered it into a file (say csx.cup), you can test whether the grammar can be parsed by a CUP-generated parser. The build file for project 3 contains a target Cup that runs Java CUP on file lite.cup. (You can edit build.xml to make target Cup process a Java CUP file other than lite.cup.) Running ant Cup

will initiate execution of Java CUP.

Alternatively, at the command line level you can enter

java java_cup.Main < csx.cup

Using the grammar we’ve provided, Java CUP will generate a message

*** Shift/Reduce conflict found in state #XX

where XX is a number that depends on the exact structure of the grammar you enter. This message indicates that the grammar we’ve provided is almost, but not quite, in a form acceptable to CUP. This is a common occurrence. Most context-free grammars used to define programming languages can be handled by CUP, sometimes after minor modification.

The difficulty in this grammar is the well-known “dangling else” problem. That is, given the statement

if (a) if (b) a=true; else b=true;

does the else statement belong to the outer or inner if? The grammar we’ve provided allows either association. The correct association is to match the else part with the nearest unmatched if. You will have to modify the grammar we’ve provided to enforce this “nearest match” rule.

Initially, you may remove the if-then production, keeping the if-then-else production. This will temporarily solve the shift/reduce conflict, allowing you to build and test a working parser.

At some point you will need to add back the if-then production. You can solve the resulting shift/reduce conflict in either of two ways. The problem is that the then-part of an if-then-else should not be allowed to generate an if-then statement. That is, using the above example, if we start with

if (a) stmt else b=true;
we have the else-part controlled by the value of variable a. But if stmt then generates

        if (b) a=true;

then it appears that variable b controls execution of the else-part. So we must rewrite the
grammar so that stmt in the then-part of an if-then-else can never generate anything that
ends with an if-then statement.

Alternatively, Java CUP allows us to use a grammar with a shift/reduce conflict if we properly
ask it to (see section 3 of the Java CUP manual (expect option)). A shift operation takes prece-
dence over a reduce operation, which (if done carefully) can correctly solve the dangling else
problem.

You may rewrite the CSX grammar in any way you wish, adding or changing productions and
nonterminals. You can't change the CSX language itself (i.e., the sequences of tokens consid-
ered valid).

Once your grammar is in the right format and generates no error messages, Java CUP will
create a file parser.java that contains the parser it has generated. It will also create a file
sym.java that contains the token codes the parser is expecting. Use sym.java with JLex in
generating your scanner to guarantee that both the scanner and parser use the same token codes.

The generated parser, named parse, is a member of class parser. It will call Scanner.
next_token() to get tokens. Class Scanner (provided by us) creates a Yylex object
(a JLex scanner) and calls yylex() as necessary to provide tokens. Be sure to call Scan-
ner.init(in) prior to parsing with in, the FileInputStream you wish to scan from.

If there is a syntax error during parsing, parse() will throw a SyntaxErrorException;
be sure to catch it. It will also call syntax_error(token) to print an error message. We pro-
vide a simple implementation of syntax_error in lite.cup (the Java CUP parser specification
for CSX-lite). You may improve it if you wish (perhaps to print the offending token). You
should test your parser on a variety of simple inputs, both legal and illegal, to verify that your
parser is operating correctly.

Generating Abstract Syntax Trees

So far your parser reads input tokens and determines whether they form a syntactically cor-
rect program. You now must extend your parser so that it builds an abstract syntax tree (AST).
The AST will be used by the type checker and code generator to complete compilation of a CSX
program.

Abstract syntax tree nodes are defined as Java classes, with each particular kind of AST
node corresponding to a particular class. Thus the AST node for an assignment statement corre-
sponds to the class asgNode. The classes comprising AST nodes are not independent. All of
them are direct or indirect subclasses of the following:

    abstract class ASTNode {
        public int    linenum;
        public int    colnum;

        ASTNode(){linenum=-1;colnum=-1;}
        ASTNode(int l,int c){linenum=l;colnum=c;}

        boolean   isNull(){return false;}  // Is this node null?

        abstract void accept(Visitor v,int indent);// Defined in sub-classes
    }
ASTNode is the base class from which all other classes for AST nodes are created. ASTNode is what is termed an abstract superclass. This means objects of this class are never created. Rather the definition serves to define the fields and methods shared by all subclasses.

ASTNode contains two instance variables: linenum and colnum. They represent the line and column numbers of the tokens from which the AST node was built. Thus for asgNode, the AST node for assignment statements, linenum and colnum would correspond to the position of the assignment’s target variable, since that’s where the assignment statement begins.

ASTNode also has two constructors that set linenum and colnum. These constructors are called by constructors of subclasses to set these two fields (to either explicit or default values).

The method isNull is used to determine if a particular AST node is “null”; that is, if it corresponds to $\lambda$. Only special “null nodes” will define their isNull function to return true; other AST nodes will inherit the definition in ASTNode.

The abstract method accept will be defined in each concrete ASTNode subclass. It acts as a “traffic cop” dispatching methods organized using the Visitor pattern (see section 7.7.2 of the class text).

An example of an AST node that we will build as a CSX program is parsed is:

```java
class classNode extends ASTNode {
    public final identNode className;
    public final memberDeclsNode members;

    classNode(identNode id, memberDeclsNode m, int line, int col){ ... }
    void accept(Visitor u, int indent) { u.visit(this, indent); }
}
```

classNode corresponds to the start symbol of all CSX programs, program. classNode is a subclass of ASTNode, so it inherits all of ASTNode’s fields and members. It contains a constructor, as all AST nodes will. This constructor sets the fields of the class. It also calls ASTNode’s constructor to set linenum and colnum. Method accept when called with a visitor class (a set of translation or analysis methods organized on a per-AST node basis) will visit the method in class $u$ defined for AST class classNode (since this is a reference to it). classNode corresponds to a non-$\lambda$ construct, so it is content to inherit and use ASTNode’s definition of isNull.

classNode contains two fields, defined as public final. These correspond to the two subtrees a classNode will contain: the name of the class (an identifier), and the declarations (fields and methods) within the class. The type declarations tell us precisely the kind of subtrees that are permitted. Thus if we tried to assign a subtree corresponding to an integer literal to className, we’d get a Java type error, because the AST node corresponding to integer literals (intLitNode) is different that the type className expects (which is identNode). The fields will not change once they are set, so they are made final.

We can now see why we’ve created so many different classes for AST nodes. Each different kind of node has its own class, and it is wrong to assign a class corresponding to one kind of AST node to a field expecting a reference to a different kind of AST node.

We list below (in Table 1) all the AST classes that we use. For each class, we list the field names in that class and the type of each field. This type will usually be a reference to a particular AST class object. In some cases a field may reference a special kind of AST node, a “null node,” that corresponds to $\lambda$. That is, if a subtree is empty, we’ll use a null node to represent that fact. For example, in a CSX class, field declarations are optional. If a class chooses to have no fields, the fields field in memberDeclsNode will point to a nullFieldDeclsNode. As you
might expect, null nodes have no internal fields. They simply serve as placeholders so that all subtrees that are expected are always present. Without null nodes, you’d have to routinely check if an AST reference is null before you use it, which is tedious and error-prone.

Some AST nodes are always leaves (e.g., `identNode`); others have one or more subtrees. Thus the `asgNode` has two subtrees, one for the name being assigned to (target) and the other for the expression being assigned (source).

The AST nodes `identNode`, `intLitNode`, `charLitNode` and `strLitNode` do not have subtrees, but do contain the string value, integer value, character value, or string value returned by the scanner (in token objects). Leaf nodes like `trueNode` and `boolTypeNode` have no fields (other than `linenum` and `colnum` inherited from their superclass). This simply means that for such nodes we need no information beyond their class.

Null nodes are used to represent null subtrees. Java’s strict type rules make it necessary to create several different classes for null nodes. However, we have made it easy to reference a null node of the correct type. If you want a null node that can be assigned to a field of class `XXX`, then `XXX.NULL` is the null node you want. For example, if you want to assign a null node to a field expecting a `stmtsOption`, then `stmtsOption.NULL` is the value you should use.

It is better to reference a null node than to store a `null` value. If all object references in AST nodes point to `something` then we never have to check a reference before we use it.

Besides `astNode`, we will use a number of other abstract superclasses to build our AST. One of these is `stmtNode`. We will never actually create a node of type `stmtNode`. But then why do we bother to define it?

Sometimes we want to be able to reference one of a number of kinds of AST nodes, but not just any node. Thus in a `stmtNode` we want to reference any kind of AST node corresponding to a statement, but not AST nodes corresponding to non-statements. We solve this problem by declaring a reference to have type `stmtNode`. We make all classes corresponding to statements (like `asgNode` or `whileNode`) subclasses of `stmtNode`. The rules of Java say that a reference to a class `S` may be assigned an object of any subclass of `S`. This is because a subclass of `S` contains everything `S` does (and perhaps more). Thus an `asgNode` may be assigned to a field expecting a `stmtNode` without error. However an AST node that is not a subclass of `stmtNode` (e.g., `boolTypeNode`) may not be legally assigned to a field expecting a `stmtNode`.

Although the set of class definitions in `ast.java` looks complex, the main benefit of using them is that it becomes very difficult to insert AST nodes in the wrong place. If you try, you’ll get an error message complaining that the type of node you are trying to assign to an AST node’s field is illegal. In Table 2, below, we list all the abstract AST nodes that appear in `ast.java` and their subclasses.

<table>
<thead>
<tr>
<th>Java class</th>
<th>Fields Used</th>
<th>Type of Fields</th>
<th>Null node allowed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classNode</code></td>
<td><code>className</code></td>
<td><code>identNode</code></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><code>members</code></td>
<td><code>memberDecsNode</code></td>
<td>No</td>
</tr>
<tr>
<td><code>memberDecsNode</code></td>
<td><code>fields</code></td>
<td><code>fieldDecsOption</code></td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td><code>methods</code></td>
<td><code>methodDecsOption</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>fieldDecsNode</code></td>
<td><code>thisField</code></td>
<td><code>declNode</code></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><code>moreFields</code></td>
<td><code>fieldDecsOption</code></td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Classes Used to Define AST Nodes in CSX
<table>
<thead>
<tr>
<th>Java class</th>
<th>Fields Used</th>
<th>Type of Fields</th>
<th>Null node allowed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>varDeclNode</td>
<td>varName</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>varType</td>
<td>typeNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>initValue</td>
<td>exprNodeOption</td>
<td>Yes</td>
</tr>
<tr>
<td>constDeclNode</td>
<td>constName</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>constValue</td>
<td>exprNode</td>
<td>No</td>
</tr>
<tr>
<td>arrayDeclNode</td>
<td>arrayName</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>elementType</td>
<td>typeNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>arraySize</td>
<td>intLitNode</td>
<td>No</td>
</tr>
<tr>
<td>intTypeNode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolTypeNode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>charTypeNode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>voidTypeNode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>methodDeclsNode</td>
<td>thisDecl</td>
<td>methodDeclNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>moreDecls</td>
<td>methodDeclsOption</td>
<td>Yes</td>
</tr>
<tr>
<td>methodDeclNode</td>
<td>name</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>args</td>
<td>argDeclNodeOption</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>returnType</td>
<td>typeNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>decls</td>
<td>fieldDeclNodeOption</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>stmts</td>
<td>stmtsNode</td>
<td>No</td>
</tr>
<tr>
<td>argDeclsNode</td>
<td>thisDecl</td>
<td>argDeclNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>moreDecls</td>
<td>argDeclNodeOption</td>
<td>Yes</td>
</tr>
<tr>
<td>arrayArgDeclNode</td>
<td>argName</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>elementType</td>
<td>typeNode</td>
<td>No</td>
</tr>
<tr>
<td>valArgDeclNode</td>
<td>argName</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>argType</td>
<td>typeNode</td>
<td>No</td>
</tr>
<tr>
<td>stmtsNode</td>
<td>thisStmt</td>
<td>stmtNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>moreStmts</td>
<td>stmtsOption</td>
<td>Yes</td>
</tr>
<tr>
<td>asgNode</td>
<td>target</td>
<td>nameNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>source</td>
<td>exprNode</td>
<td>No</td>
</tr>
<tr>
<td>incrementNode</td>
<td>target</td>
<td>nameNode</td>
<td>No</td>
</tr>
<tr>
<td>decrementNode</td>
<td>target</td>
<td>nameNode</td>
<td>No</td>
</tr>
<tr>
<td>ifThenNode</td>
<td>condition</td>
<td>exprNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>thenPart</td>
<td>stmtNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>elsePart</td>
<td>stmtOption</td>
<td>Yes</td>
</tr>
<tr>
<td>whileNode</td>
<td>label</td>
<td>exprNodeOption</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>condition</td>
<td>exprNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>loopBody</td>
<td>stmtNode</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1.  Classes Used to Define AST Nodes in CSX
### Table 1. Classes Used to Define AST Nodes in CSX

<table>
<thead>
<tr>
<th>Java class</th>
<th>Fields Used</th>
<th>Type of Fields</th>
<th>Null node allowed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>readNode</td>
<td>targetVar, moreReads</td>
<td>nameNode, readNodeOption</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>printNode</td>
<td>outputValue, morePrints</td>
<td>exprNode, printNodeOption</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>callNode</td>
<td>methodName, args</td>
<td>identNode, argsNodeOption</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>returnNode</td>
<td>returnVal, morePrints</td>
<td>exprNodeOption</td>
<td>Yes</td>
</tr>
<tr>
<td>breakNode</td>
<td>label, morePrints</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td>continueNode</td>
<td>label, morePrints</td>
<td>identNode</td>
<td>No</td>
</tr>
<tr>
<td>blockNode</td>
<td>decls, stmts</td>
<td>fieldDecllsOption, stmtsOption</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>argsNode</td>
<td>argVal, moreArgs</td>
<td>exprNode, argsNodeOption</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>strLitNode</td>
<td>strval</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>binaryOpNode</td>
<td>leftOperand, rightOperand</td>
<td>exprNode, exprNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>unaryOpNode</td>
<td>operand, operatorCode</td>
<td>exprNode, int</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>castNode</td>
<td>resultType, operand</td>
<td>typeNode, exprNode</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>fctCallNode</td>
<td>methodName, methodArgs</td>
<td>identNode, argsNodeOption</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>identNode</td>
<td>idname</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>nameNode</td>
<td>varName, subscriptVal</td>
<td>identNode, exprNodeOption</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>intLitNode</td>
<td>intval</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>charLitNode</td>
<td>charval</td>
<td>char</td>
<td>No</td>
</tr>
<tr>
<td>trueNode</td>
<td>none</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>falseNode</td>
<td>none</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>null nodes</td>
<td>none</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>(many kinds)</td>
<td></td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

**Building ASTs in Java CUP**

We’ll need to build ASTs for CSX programs we have parsed. One of the reasons we’re using Java CUP to build our parser is that it’s easy to build ASTs using CUP. CUP allows us to embed actions, in the form of Java code, in the productions CUP parses. When a production containing
an action is matched by `parse()`, the associated action is automatically executed. For example
in the following rule (drawn from `lite.cup`)

```
stmt ::= LBRACE:l fielddecls:f stmts:s RBRACE optionalSemi
{: RESULT=new blockNode(f,s, l.linenum, l.colnum);
:}
```

the production `stmt → { fielddecls stmts } optionalSemi` is specified. Moreover, whenever
this production is matched, the constructor `blockNode` is called (since `blockNode` corre-
sponds to block statements). The constructor for `blockNode` wants four things: ASTs nodes

<table>
<thead>
<tr>
<th>Abstract AST Node</th>
<th>Subclasses</th>
<th>Abstract AST Node</th>
<th>Subclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>argDeclNode</td>
<td>arrayArgDeclNode, valArgDeclNode</td>
<td>argDeclsOption</td>
<td>argDeclsNode, nullArgDeclsNode</td>
</tr>
<tr>
<td>argsNodeOption</td>
<td>argsNode, nullArgsNode</td>
<td>declNode</td>
<td>varDeclNode, constDeclNode, arrayDeclNode</td>
</tr>
<tr>
<td>exprNode</td>
<td>binaryOpNode, castNode, charLitNode, falseNode, fctCallNode, identNode, intLitNode, nameNode, strLitNode, trueNode, unaryOpNode</td>
<td>stmtNode</td>
<td>asgNode, blockNode, breakNode, callNode, continueNode, ifThenNode, printNodeOption, readNodeOption, returnNode, whileNode, incrementNode, decrementNode</td>
</tr>
<tr>
<td>exprNodeOption</td>
<td>exprNode, nullExprNode</td>
<td>fieldDeclsOption</td>
<td>fieldDeclsNode, nullFieldDeclsNode</td>
</tr>
<tr>
<td>methodDeclsOption</td>
<td>methodDeclsNode, nullMethodDeclsNode</td>
<td>printNodeOption</td>
<td>printNode, nullPrintNode</td>
</tr>
<tr>
<td>readNodeOption</td>
<td>readNode, nullReadNode</td>
<td>stmtOption</td>
<td>stmtNode, nullStmtNode</td>
</tr>
<tr>
<td>stmtsOption</td>
<td>stmtsNode, nullStmtsNode</td>
<td>typeNode</td>
<td>boolTypeNode, charTypeNode, intTypeNode, voidTypeNode</td>
</tr>
<tr>
<td>typeNodeOption</td>
<td>typeNode, nullTypeNode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2  Abstract Classes Used in AST Nodes and Their Subclasses
corresponding to the declarations and statements in the block, and a line and column number to associate with the block. The special suffixes :l, :f and :s represent references (automatically maintained by CUP) to the tokens and ASTs for the {, fielddecls and stmts that have already been parsed. The ASTs have already been built by the time this production is matched. We define the line and column of the block to be the line and column of the leftmost symbol in the block, which is the {Since, l references the token corresponding to {, l.linenum represents the line number already stored for the {.

After blockNode builds a new AST node for the block and links in its subtrees, the result is assigned to RESULT. RESULT is a special symbol, maintained by CUP, that represents the left-hand side non-terminal (stmt). As productions are matched, AST subtrees are built and merged into progressively larger trees. Finally, when the first production (corresponding to an entire program) is matched, the root of the complete AST can be returned by the parser. The bookkeeping required to maintain AST pointers as productions are matched is automatically done by CUP, using the RESULT and :name notation.

The objects referenced for each terminal and non-terminal symbol in the grammar are defined using terminal and non terminal directives. The lines

```
terminal CSXIdentifierToken     IDENTIFIER;
terminal CSXToken SEMI, LPAREN, RPAREN, ASG, LBRACE, RBRACE;
```

tell Java CUP that the tokens for ';', '(', ')', etc. will all be instances of class CSXToken, while the IDENTIFIER token will be an instance of class CSXIdentifierToken. The lines

```
non terminal csxLiteNode        prog;
non terminal stmtsOption       stmts;
```
say that the nonterminal prog will reference class csxLiteNode, while the nonterminal stmts will reference stmtsOption.

The member function parse(), which is the CUP-generated parser, returns an object of type Symbol. For successful parses, this will be the start symbol (program) of the derivation. The value field of the returned Symbol will contain the AST corresponding to program.

**Unparsing**

For grading, testing and debugging purposes, it is necessary to display the abstract syntax tree your parser creates. A convenient way to do this is to create a collection of “unparsing methods,” one for each kind of AST node. The natural place to locate these methods in within AST node classes. But this approach is problematic. Many components of our compiler operate by traversing the AST. If each compiler phase places its methods within AST classes, the classes soon become large and unreadable, cluttered with methods for many different analyses and translations.

An alternative is to use the “visitor pattern” (see section 7.7.2 of the class text). All the methods needed to implement a given compiler phase (like unparsing) are placed in a single class that is derived from class Visitor. Hence we put all unparsing methods (one for each kind of AST node) in class Unparsing, a subclass of Visitor. We start by calling the unparsing method corresponding to the root of the AST (classNode) in the main method of P3. The unparsing method for classNode can then call unparsing methods for its subtrees, unparsing their content. This works nicely until we need to unpars a subtree for which we don’t know the exact type of the root. This is common, since we often define the type of a subtree to be an abstract class.
Here the cleverness of the visitor pattern becomes evident. In the `Visitor` class we have a special definition of `visit`, used by all classes (particularly abstract classes) that don’t provide their own definition of `visit`:

```java
public void visit(ASTNode n, int indent){
    n.accept(this, indent); }
```

This method calls an `accept` method for node `n`, an AST node. Each concrete (non-abstract) AST node has a definition of `accept`. In all cases this definition is the same! It is

```java
void accept(Visitor u, int indent){ u.visit(this,indent); }
```

The `accept` in the exact class we want to visit is executed. Thus in the call `u.visit(this,indent)` we execute the `visit` method corresponding to `this`, which has the exact AST node type we want to process.

Each `visit` method in class `Unparsing` prints out the structure of some AST node in conventional (text-oriented) form. (The parameter `indent` is the number of tabs to indent prior to printing the node’s structure.) Unparsing methods “pretty print” a construct, adding new lines and tabs as appropriate to create a pleasing and easily-readable listing. For constructs that are forced to begin on a new line (like statements and declarations) you should print a line number at the beginning of the construct’s unparsing using the `linenum` value stored in the AST node. Note that the line numbers printed may not be consecutive since they correspond to the original input text. Moreover, some parts of a construct that appear on a new line (like the ‘}’ at the end of the class definition) will get a line number that appears “out of order” since the line number stored with an AST node corresponds to where the construct began.

Each abstract syntax tree node is associated with a production that can be viewed as a pattern that specifies how a node is to be displayed. For example assume we must unparsen an `asgNode`. It’s unparsing method, in class `Unparsing`, is

```java
void visit(asgNode n, int indent){
    System.out.print(n.linenum + ":");
    genIndent(indent);
    this.visit(n.target,0);
    System.out.print(" = ");
    this.visit(n.source,0);
    System.out.println(";");
}
```

An assignment is always be printed on a new line, so we first print out the line number (using the node’s `linenum` value) and indent using the `indent` parameter. We then call `this.visit(n.target,0)`. This executes the `visit` method in class `Unparsing` for `n.target`, which is a `nameNode`. This call prints the target variable, without indenting. Next we print ‘=’, and then call `this.visit(n.source,0)` to print the source expression, without indenting. Finally, we print ‘;’.

To unparsen `intLitNodes` we print `intval`. For `strLitNodes` we print `strval` (which is the full string representation, with quotes and escapes). For `charLitNodes` we print `charval` as a quoted character in fully escaped form. For `identNodes` we use `idname` which is the text of the identifier.

Abstract syntax trees for expressions contain no parentheses since the tree structure encodes how operands are grouped. When expressions are unparsed, explicit parentheses should be added to guarantee that expressions are properly interpreted. Hence `A+B*C` would be unparsed as `(A+(B*C))`. (Fancier unparsers that only print necessary parentheses are a bit harder to write. An unparser that prints parentheses only when really necessary will get extra credit.)
What You Must Do

This project step is not nearly as hard as it looks, because you have CUP to help you build your parser. Still, it helps a lot to see an example of all the pieces you’ll need to complete. We’ve created a small subset of CSX, called CSX-lite, that’s defined by the following productions:

```
program → { fielddecls stmts }
fielddecls → fielddecl fielddecls
           | λ
fielddecl → type id ;
type → int | bool
stmts → stmt stmts
       | λ
stmt → id = expr ;
    | if ( expr ) stmt ;
    | { fielddecls stmts } optionalSemi
expr → expr + unit
    | expr - unit
    | expr == unit
    | expr != unit
unit → ( expr )
    | intlit
    | id
optionalSemi → ;
              | λ
```

CSX-lite Grammar

This subset contains only simple variable declarations. The only statements are assignment, conditional (if statements) and blocks. Expressions involve only +, −, ==, and !=, as well as identifiers and integer literals. Complete CUP specifications, parsers, AST builders and unparsers for CSX-lite may be found at the class web page (Programming Assignments section) as an Eclipse archive file (cs 536 project 3.zip). The material is also available in a folder (Java Code) at the same web page.

You should look at what we’ve provided to make sure you understand how each step of the project works for CSX-lite. Basically, ASTs are built using calls to constructors as illustrated in lite.cup. Once an individual production is matched by the parser, a constructor for the corresponding AST node is called. You should substitute your scanner from project 2, by replacing lite.jlex with your csx.jlex file. (Be sure to update the build.xml file with the name of the new JLex file.)

Unparsing functions, one for each AST node that is built, are in Unparsing.java. We’ve created prototypes for each unparsers you’ll need. Replace implementations that warn of “not implemented yet” with appropriate unparsing actions.

Once you’re clear on what’s going on, add a single simple feature like a variable declaration or a while loop. This involves first adding the appropriate productions to the CUP specification. Build the parser and verify that you get no syntax errors when you parse source files containing the new construct. Next, add constructor actions to your CUP specification to build ASTs for the construct you’ve added. Then complete unparsing methods for the nodes you’ve built. Now you can verify the ASTs you built are correct by looking at the unparsing you generate. To aid in test-
ing, we’ve added a new target called test1 in build.xml. If you use this target, your source files will be recompiled as necessary, and then you will be prompted to enter the name of a test file. In the simpler form (the default) a box requesting a file name will appear. A more complex version creates a “file chooser” box that lists available files and allows you to click on the file you want to use. To activate this version, change the constant SIMPLE_GUI (in source file ArgsProcessor.java) from true to false.

After you have added a few constructs, you should have a good understanding of all the steps involved. Then you can incrementally add the complete set of CSX productions to your CUP specification, eventually creating a complete CSX parser and unparser.

Error Handling

In the case of syntax errors CUP will call syntax_error() to print an error message and then throw a SyntaxErrorException, indicating abnormal termination. The caller of your parser should catch this exception, which indicates that because of errors no AST could be built.

CUP does provide a simple error recovery mechanism (using “error” markers). This is described in §5 of the CUP manual. If you wish, you may experiment with syntactic error recovery after your parser is fully operational.

What to Hand In

As input, your parser will take a text file on the command line, which will be passed to the scanner to read and build tokens for the parser (if no file name is found on the command line, a GUI will prompt you to enter one). You should test your parser on syntactically valid and invalid programs. For invalid programs, your error messages should be clear and meaningful. For valid programs, you should show a readable unparsed listing of the abstract syntax tree that is created.

Create a folder (directory) and name it using your first and last name (e.g., CharlesFischer). Copy into this folder a README file, a build.xml file and all source files necessary to build an executable version of your program (.java source files, a csx.jlex file and a csx.cup file). Do not hand in any .class files. Name the class that contains your main P3.java. Upload this handin folder to the cs 536 project Dropbox folder in learn@uw (https://learnuw.wisc.edu). You may compress your handin folder into a single file using zip if you wish. Partners should submit only one solution. The other partner should submit only a README file identifying the partnership.

If you wish to claim extra credit, clearly state (in the README file) what you’ve added and include examples of its operation.