CS 536 — Spring 2006

Programming Assignment 3
CSX Parser

Due: Wednesday, March 22, 2006

Not accepted after Wednesday, March 29, 2006

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 precedence. Thus `!A+B*C==3 || D!=F` is equivalent to the following fully-parenthesized expression: `((((!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 operators are (of course) right-associative. Be sure that your parser for CSX properly reflects these precedence and associativity rules.

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
	|	stmt
stmt	→	if (expr) stmt

```
                |    if     (  expr  )   stmt   else   stmt
                |    while  (  expr  )   stmt
                |    id  :  while  (  expr  )   stmt
                |    name   =    expr   ;
                |    read   (  readlist  )   ;
                |    print   (  writelist   )   ;
                |    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
writelist       →    expr    ,   writelist
                |    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
                |    charlit
                |    strlit
                |    true
                |    false
                |    (    expr    )
```

name	$\longrightarrow$	id				
	|	id	[	expr	]	

CSX Grammar

Using JavaCUP to Build a Parser

You will use *JavaCUP*, a Java-based parser generator, to build your CSX parser. You'll have to rewrite the CSX grammar into the format required by JavaCUP. This format is defined in "CUP User's Manual," available in the "Useful Programming Tools" section of the class homepage. A sample CUP specification corresponding to *CSX-lite* (a small subset of CSX) may be found in `~cs536-1/public/proj3/startup/lite.cup`.

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. Run

```
java java_cup.Main < csx.cup
```

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 must modify the grammar we've provided to enforce this "nearest match" rule. See §5.6 of the compiler notes for a more thorough discussion of the problem (CUP generates LALR(1) parsers, so a correct grammar *can* be written.)

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 considered 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 `Scanner.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 provide 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 correct 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 corresponds 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 {
    int     linenum;
    int     colnum;

    static void genIndent(int indent){ ... }

    ASTNode(){linenum=-1;colnum=-1;}
    ASTNode(int l,int c){linenum=l;colnum=c;}
    boolean   isNull(){return false;}; // Is this node null?
    void Unparse(int indent){};
};
```

`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 λ. Only special "null nodes" will define their `isNull` function to return true; other AST nodes will inherit the definition in `ASTNode`.

The method `Unparse` is used to "unparse" an AST node—that is, to print it out in a clear human-readable form. Unparsing will be discussed below. We expect that each subclass will provide its own definition of `Unparse`; the default—to print nothing—is usually inappropriate. Thus the `asgNode`'s `Unparse` function will define how assignment statements are to be printed. Clearly each kind of AST node should have its own unparsing rules. Member `genIndent` is a utility routine used by `Unparse`.

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

```
 class classNode extends ASTNode {
       classNode(identNode id, memberDeclsNode m,
                               int line,int col){ ... }
       void Unparse(int indent) { ... }
       private identNode       className;
       private memberDeclsNode members;
};
```

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 private members of the class. It also calls ASTNode's constructor to set linenum and colnum. Unparse provides a definition of unparsing appropriate to the program structure the class represents. Since classNode corresponds to a non-λ construct, it is content to inherit and use ASTNode's definition of isNull.

classNode also contains two private fields. 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).

This explains 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 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 λ. That is, if a subtree is empty, we'll use a null node to represent that fact. For example, in a class method declarations are optional. If a class chooses to have no methods, the methods 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, it is 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 stmtNode, then stmtNode.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 readNode) *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 variable 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 variable 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 AST nodes that appears in `ast.java` and their superclass.

Java class	**Fields Used**	**Type of Fields**	**Null node allowed?**
`classNode`	`className`	`identNode`	No
	`members`	`memberDeclsNode`	Yes
`memberDeclsNode`	`fields`	`fieldDeclsNode`	Yes
	`methods`	`methodDeclsNode`	Yes
`fieldDeclsNode`	`thisField`	`declNode`	No
	`moreFields`	`fieldDeclsNode`	Yes
`varDeclNode`	`varName`	`identNode`	No
	`varType`	`typeNode`	No
	`initValue`	`exprNode`	Yes
`constDeclNode`	`constName`	`identNode`	No
	`constValue`	`exprNode`	No
`arrayDeclNode`	`arrayName`	`identNode`	No
	`elementType`	`typeNode`	No
	`arraySize`	`intLitNode`	No
`intTypeNode`			
`boolTypeNode`			
`charTypeNode`			
`voidTypeNode`			
`methodDeclsNode`	`thisDecl`	`methodDeclNode`	No
	`moreDecls`	`methodDeclsNode`	Yes
`methodDeclNode`	`name`	`identNode`	No
	`args`	`argDeclsNode`	Yes
	`returnType`	`typeNode`	No
	`decls`	`fieldDeclsNode`	Yes
	`stmts`	`stmtsNode`	No
`argDeclsNode`	`thisDecl`	`argDeclNode`	No
	`moreDecls`	`argDeclsNode`	Yes
`arrayArgDeclNode`	`argName`	`identNode`	No
	`elementType`	`typeNode`	No

Table 1. Classes Used to Define AST Nodes in CSX

Java class	Fields Used	Type of Fields	Null node allowed?
`valArgDeclNode`	`argName`	`identNode`	No
	`argType`	`typeNode`	No
`stmtsNode`	`thisStmt`	`stmtNode`	No
	`moreStmts`	`stmtsNode`	Yes
`asgNode`	`target`	`nameNode`	No
	`source`	`exprNode`	No
`ifThenNode`	`condition`	`exprNode`	No
	`thenPart`	`stmtNode`	No
	`elsePart`	`stmtNode`	Yes
`whileNode`	`label`	`exprNode`	Yes
	`condition`	`exprNode`	No
	`loopBody`	`stmtNode`	No
`readNode`	`targetVar`	`nameNode`	No
	`moreReads`	`readNode`	Yes
`printNode`	`outputValue`	`exprNode`	No
	`morePrints`	`printNode`	Yes
`callNode`	`methodName`	`identNode`	No
	`args`	`argsNode`	Yes
`returnNode`	`returnVal`	`exprNode`	Yes
`breakNode`	`label`	`identNode`	No
`continueNode`	`label`	`identNode`	No
`blockNode`	`decls`	`fieldDeclsNode`	Yes
	`stmts`	`stmtsNode`	No
`argsNode`	`argVal`	`exprNode`	No
	`moreArgs`	`argsNode`	Yes
`strLitNode`	`strval`	`String`	No
`binaryOpNode`	`leftOperand`	`exprNode`	No
	`rightOperand`	`exprNode`	No
	`operatorCode`	`int`	No
`unaryOpNode`	`operand`	`exprNode`	No
	`operatorCode`	`int`	No
`castNode`	`resultType`	`typeNode`	No
	`operand`	`exprNode`	No
`fctCallNode`	`methodName`	`identNode`	No
	`methodArgs`	`argsNode`	Yes
`identNode`	`idname`	`String`	No
`nameNode`	`varName`	`identNode`	No
	`subscriptVal`	`exprNode`	Yes

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

Java class	Fields Used	Type of Fields	Null node allowed?
`intLitNode`	`intval`	`int`	No
`charLitNode`	`charval`	`char`	No
`trueNode`	`none`		
`falseNode`	`none`		
`null nodes (many kinds)`	`none`		

Table 1. Classes Used to Define AST Nodes in CSX

AST Node	Superclass	AST Node	Superclass
`argDeclNode`	`ASTNode`	`argDeclsNode`	`ASTNode`
`argsNode`	`ASTNode`	`arrayArgDeclNode`	`argDeclNode`
`arrayDeclNode`	`declNode`	`asgNode`	`stmtNode`
`binaryOpNode`	`exprNode`	`blockNode`	`stmtNode`
`boolTypeNode`	`typeNode`	`breakNode`	`stmtNode`
`callNode`	`stmtNode`	`castNode`	`exprNode`
`charLitNode`	`exprNode`	`charTypeNode`	`typeNode`
`classNode`	`ASTNode`	`constDeclNode`	`declNode`
`continueNode`	`stmtNode`	`declNode`	`ASTNode`
`exprNode`	`ASTNode`	`falseNode`	`exprNode`
`fctCallNode`	`exprNode`	`fieldDeclsNode`	`ASTNode`
`identNode`	`exprNode`	`ifThenNode`	`stmtNode`
`intLitNode`	`exprNode`	`intTypeNode`	`typeNode`
`memberDeclsNode`	`ASTNode`	`methodDeclNode`	`ASTNode`
`methodDeclsNode`	`ASTNode`	`nameNode`	`exprNode`
`nullNode`	`ASTNode`	`printNode`	`stmtNode`
`readNode`	`stmtNode`	`returnNode`	`stmtNode`
`stmtNode`	`ASTNode`	`stmtsNode`	`ASTNode`
`strLitNode`	`exprNode`	`trueNode`	`exprNode`
`typeNode`	`ASTNode`	`unaryOpNode`	`exprNode`
`valArgDeclNode`	`argDeclNode`	`varDeclNode`	`declNode`
`whileNode`	`stmtNode`	`voidTypeNode`	`typeNode`

Table 2 Classes Used in AST Nodes and Their Superclasses

Getting Started

We've placed skeleton files for the project in `~cs536-1/public/proj3/startup`. Look at file `ast.java`. This file will create a large number of ".class" files (one for each kind of AST node, as well as others). To keep your project directory manageable, the `Makefile` places all ".class" files in a subdirectory, `classes`. Be sure your `CLASSPATH` environment variable to include this directory.

If you haven't already done so, update your `.cshrc.local` file (which can be found in your home directory) to contain:

```
setenv  CLASSPATH ".:./classes:/s/java/jre/lib/rt.jar:/p/course/
cs536-reps/public/JAVA"
```

```
setenv VPATH "./classes"
```

(These are two lines, not three. Ignore the line break after the `/`. This lines are exactly the same as we used in projects 1 and 2)

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 ::= ident:id   ASG   exp:e   SEMI
    {: RESULT =
       new asgNode(id, e, id.linenum, id.colnum);
    :}
```

the production stmt → ident = expr ; is specified. Moreover, whenever this production is matched, the constructor `asgNode` is called (since `asgNode` corresponds to assignment statements). The constructor for `asgNode` wants four things: ASTs nodes corresponding to the source and target of the assignment, and a line and column number to associate with the assignment. The special suffixes `:id` and `:e` represent references (automatically maintained by CUP) to the ASTs for the ident and expr that have already been parsed. These ASTs have already been built by the time this production is matched. We define the line and column of the assignment to be the line and column of the leftmost symbol in the assignment, which is the ident. Since, `id` references the AST node built for ident, `id.linenum` represents the line number already stored for the identifier.

After `astNode` builds a new AST node for the assignment and links in its subtrees, the result is assigned to `RESULT`. `RESULT` is a special symbol, maintained by CUP, that represents the lefthand 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.

Information placed in tokens returned by the scanner can also be easily accessed. A suffix placed after a terminal symbol allows the token object corresponding to the terminal symbol to be accessed. Thus the rule

```
exp ::= exp:l    PLUS:op    ident:r
    {: RESULT = new binaryOpNode(l, sym.PLUS, r,
                                     op.linenum, op.colnum); :}
```

uses the `linenum` and `colnum` values of the `PLUS` token (extracted as `op.linenum` and `op.colnum`) in constructing a `binaryOpNode` that represents the AST for the addition operation.

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 stmtsNode          stmts;
```

say that the nonterminal `prog` will reference class `csxLiteNode`, while the nonterminal `stmts` will reference `stmtsNode`.

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 member function `Unparse(int indent)` that prints out the node's structure in conventional (text-oriented) form. (`indent` is the number of tabs to indent prior to printing the node's structure.) `Unparse` "pretty prints" the 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 given an `asgNode`, which will always be printed on a new line, we would first print out the line number (using the node's `linenum` value) and indent using `Unparse`'s `indent` parameter. We then call `target.Unparse(0)` (to print the target variable, without indenting), print '=', call `source.Unparse(0)` (to print the source expression, without indenting), and finally print '`;`'.

For `intLitNode`s we print `intval`. For `strLitNode`s we print `strval` (which is the full string representation, with quotes and escapes). For `charLitNode`s print `charval` as a quoted character in fully escaped form. For `identNode`s the unparser should 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 to `(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	$\longrightarrow$	{ stmts }
stmts	$\longrightarrow$	stmt stmts
	|	λ
stmt	$\longrightarrow$	id = expr ;
	|	if (expr) stmt
expr	$\longrightarrow$	expr + id
	|	expr - id
	|	id

CSX-lite Grammar

This simple subset contains no declarations; only an assignment and if statement are provided and expressions involve only `+`, – and identifiers. Complete CUP specifications, parsers, AST builders and unparsers for CSX-lite may be found in `~cs536-1/public/proj3/startup`. Just type

```
make test
```

to build a complete parser for CSX-lite and then test it using a simple source program.

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.

Unparsing functions, one for each AST node that is built, are member functions in `ast.java`. Each such routine is fairly simple—the information in the node is printed in nicely formatted form, with recursive calls to `Unparse` to unparse subcomponents.

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 define `Unparse` in the nodes you've built to unparse ASTs for this construct. Now you should be able to verify the ASTs you built are correct by looking at the unparsing you generate.

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. 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. Hand in a listing of your parser module and your CUP specification and a listing of your parser's execution on a variety of syntactically valid and invalid programs.

We've created a directory for you using your login in `~cs536-1/public/proj3/handin`. Copy into your handin directory a `README` file, a `Makefile` (if you changed the file we provided), and all source files (`.cup`, `.jlex` and `.java` files) necessary to build an executable version of your program. Do not hand in executables or `.class` files. Name the class that contains your `main` `P3.java`.

If you wish to claim extra credit, *clearly* state what you've added and include examples of its operation.