CS 541 — Fall 2020

Programming Assignment 3
CSX_go Parser

Write a Java CUP parser specification to implement a CSX_go parser. A grammar that defines CSX_go syntax appears below. You should examine the grammar carefully to learn the structure of CSX_go constructs. In most cases, structures are very similar to those of Java and C++. At this stage, you need not understand exactly what each construct *does*, but rather just how each construct *appears*. Labeled **`for`** statements and the **`break`** and **`continue`** statements are optional; you get extra credit if you implement them.

The CSX_go grammar listed below encodes the fact that the unary `!` Operator has 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 (for instance, `A==B==C` is invalid). The unary operators are (of course) right-associative. Be sure that your parser for CSX_go properly reflects these precedence and associativity rules.

```
program → package identifier varDecls funcDecls

varDecls → varDecl varDecls
        | λ

varDecl → var   identifier type ;
        | var   identifier = expr ;
        | var   identifier type [ intlit ] ;
        | const identifier = expr ;

funcDecls → funcDecl funcDecls
    | λ

funcDecl → func identifier ( formals ) optType block

formals → someFormals
        | λ
```

```
someFormals → formalDecl
            | formalDecl , someFormals

formalDecl → identifier type
           | identifier [ ] type

optType → type
        | λ

stmts → stmt stmts
      | λ

stmt → if expr block
     | if expr block else block
     | for expr block
     | identifier : for expr block
     | name = expr ;
     | read readlist ;
     | print printlist ;
     | identifier ( actuals ) ;
     | return ;
     | return expr ;
     | break identifier ;
     | continue identifier ;
     | block

block → { vardecls stmts } optionalSemi

optionalSemi → ;
             | λ

type → int
     | char
     | bool

actuals → someActuals
        | λ

someActuals → expr
            | expr , someActuals

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 ( expr )
      | unit

unit → name
     | identifier ( actuals )
     | intlit
     | charlit
     | strlit
     | boollit
     | ( expr )

name → identifier
     | identifier [ expr ]
```

CSX_go Grammar

Using JavaCUP to Build a Parser

You will use *JavaCUP*, a Java-based parser generator, to build your CSX_go parser. You'll have to rewrite the CSX_go grammar into the format required by JavaCUP. This format is defined in the "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_go) may be found in
`~raphael/courses/cs541/public/proj3/startup/csx_lite.cup`.

Once you've rewritten the CSX_go grammar we've provided and entered it into a file (say `CSX_go.cup`), you can test whether the grammar can be parsed by a CUP-generated parser. Run

```
java java_cup.Main < CSX_go.cup
```

Java CUP might 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 problem is a common occurrence. Most context-free grammars that are used to define programming languages can be handled by *JavaCUP*, sometimes after minor modification.

You may rewrite the CSX_go grammar in any way you wish, adding or changing productions and nonterminals. You **must not** change the CSX_go language itself (the sequences of tokens considered valid).

Once your grammar is in the right format and generates no error messages, *JavaCUP* creates a file `parser.java` that contains the parser it has generated. It also creates a file `sym.java`, which 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 tables are in `parser.java`. Compiling this file generates some warnings that you may ignore. It calls `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 `Reader` you wish to scan from.

If there is a syntax error during parsing, `parse()` throws a `java.lang.Exception`; be sure to catch it. It also calls `syntax_error(token)` to print an error message. We provide a simple implementation of `syntax_error` in `lite.cup` (the 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 valid and invalid, to verify that your parser is operating correctly.

Generating Abstract Syntax Trees

You should consider the material in this section a hint, not a requirement.

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 in the next projects by the type checker and code generator to complete compiling a CSX_go program.

Abstract syntax tree nodes are defined as Java classes, with each particular kind of AST node corresponding to a particular class. 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 descend. `ASTNode` is an *abstract superclass;* objects of this class are never created. Its 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` record 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" define their isNull function to return true; other AST nodes 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 is discussed below. Each subclass provides its own definition of unparse(); the default — to print nothing — is usually inappropriate. Thus the AsgNode's unparse() defines how assignment statements are to be printed. 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 we might build while parsing a CSX_go program is:

```
 class ProgramNode extends ASTNode {
       ProgramNode(IdentNode id, MemberDeclsNode m,
             int line, int col){ ... }
       void unparse(int indent) { ... }
       private IdentNode packageName;
       private List<VarDeclNode> varList;
       private List<FuncDeclNode> funcList;
};
```

ProgramNode corresponds to the start symbol of all CSX_go programs, program. ProgramNode is a subclass of ASTNode, so it inherits all of ASTNode's fields and members. It contains a constructor, as do all AST nodes. This constructor sets the private members of the class. It also calls ASTNode's constructor to set lineNum and colNum. Its unparse() provides a definition of unparsing appropriate to the program structure the class represents. Since ProgramNode corresponds to a non-λ construct, it is content to inherit and use ASTNode's definition of isNull.

ProgramNode also contains three private fields, which correspond to the subtrees a ProgramNode contains: the name of the project (an identifier), and the declarations (variables and functions) within the program. The type declarations tell us *precisely* the kind of subtrees that are permitted. If we try to assign a subtree corresponding to an integer literal to className, we get a Java type error, because the AST node corresponding to integer literals (IntLitNode) is different from the type that packageName expects (which is IdentNode).

This precaution 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 you might use. For each class, we list the field names in that class and the type of each field. This type is usually 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 use a null node to represent that fact. For example, in a function, declarations are optional. 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. Null nodes 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.

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). For such nodes, we need no information beyond their class.

Besides `astNode`, we use a number of other abstract superclasses to build our AST. One of these is `StmtNode`. We 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`. 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 assigned to a variable expecting a `StmtNode`.

Although the set of suggested 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 invalid. In Table 2, below, we list all the AST nodes that might appear in `ast.java` and their superclass.

Table 1. **Suggested classes for AST Nodes in CSX_go**

Java class	**Fields Used**	**Type of Fields**	**Null node allowed?**
`ProgramNode`	`packageName`	`IdentNode`	No
	`varList`	`List<VarDeclNode>`	No
	`funcList`	`List<FuncDeclNode>`	No
`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`			

FuncDeclNode	name args returnType body	IdentNode List<ArgDeclNode> TypeNode BlockNode	No Yes No No
ArrayArgDeclNode	argName elementType	IdentNode TypeNode	No No
ValArgDeclNode	argName argType	IdentNode TypeNode	No No
AsgNode	target source	NameNode ExprNode	No No
IfThenNode	condition thenPart elsePart	ExprNode BlockNode BlockNode	No No Yes
ForNode	label condition loopBody	IdentNode ExprNode BlockNode	Yes No No
ReadNode	targetVar moreReads	NameNode ReadNode	No Yes
PrintNode	outputValue moreDisplays	ExprNode PrintNode	No Yes
CallNode	methodName args	IdentNode ArgsNode	No Yes
ReturnNode	returnVal	ExprNode	Yes
BreakNode	label	IdentNode	No
ContinueNode	label	IdentNode	No
BlockNode	decls stmts	List<VarDeclNode> List<StatementNode>	Yes No
ArgsNode	argVal moreArgs	ExprNode ArgsNode	No Yes
StrLitNode	strval	String	No
BinaryOpNode	leftOperand rightOperand operatorCode	ExprNode ExprNode int	No No No
UnaryOpNode	operand operatorCode	ExprNode int	No No
CastNode	resultType operand	TypeNode ExprNode	No No
FuncCallNode	funcName funcArgs	IdentNode ArgsNode	No Yes
IdentNode	idname	String	No

NameNode	varName subscriptVal	IdentNode ExprNode	No Yes
IntLitNode	intval	int	No
CharLitNode	charval	char	No
TrueNode	none		
FalseNode	none		
null nodes (many kinds)	none		

Table 2 Classes Used in AST Nodes and Their Superclasses

AST Node	Superclass	AST Node	Superclass
ArgDeclNode	ASTNode	VoidTypeNode	TypeNode
ArgsNode	ASTNode	ArrayArgDeclNode	ArgDeclNode
ArrayDeclNode	DeclNode	AsgNode	StmtNode
BinaryOpNode	ExprNode	BlockNode	StmtNode
BoolTypeNode	TypeNode	BreakNode	StmtNode
CallNode	StmtNode	CastNode	ExprNode
CharLitNode	ExprNode	CharTypeNode	TypeNode
ProgramNode	ASTNode	ConstDeclNode	DeclNode
ContinueNode	StmtNode	FuncDeclNode	DeclNode
ExprNode	ASTNode	FalseNode	ExprNode
FuncCallNode	ExprNode	VarDeclNode	DeclNode
IdentNode	ExprNode	IfThenNode	StmtNode
IntLitNode	ExprNode	IntTypeNode	TypeNode
ForNode	StmtNode	UnaryOpNode	ExprNode
ValArgDeclNode	ArgDeclNode	NameNode	ExprNode
nullNode	ASTNode	PrintNode	StmtNode
ReadNode	StmtNode	ReturnNode	StmtNode
StmtNode	ASTNode	StmtsNode	ASTNode
StrLitNode	ExprNode	TrueNode	ExprNode
TypeNode	ASTNode		

Getting Started

We've placed skeleton files for the project in `~raphael/courses/cs541/public/proj3/startup`. Look at file `ast.java`. This file compiles to 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 includes this directory. The Java code in the skeleton files is not up to standard; you should use a style checker to improve the code.

Building ASTs in Java CUP

We'll need to build ASTs for CSX_go programs we have parsed. One of the reasons we're using *JavaCUP* to build our parser is that it's easy to build ASTs. *JavaCUP* allows us to embed **actions**, in the form of Java code, in the productions *JavaCup* parses. When `parse()` matches a production containing an action it automatically executes that action. For example the following rule (drawn from `lite.cup`)

```
stmt ::= ident:id  ASG  exp:e  SEMI
    {: RESULT =
       new AsgNode(id, e, id.lineNum, id.colNum);
    :}
```

specifies the production `stmt → ident = expr ;` . Whenever `parse()` matches this production, it calls the constructor `AsgNode` (since `AsgNode` corresponds to assignment statements). The constructor for `AsgNode` takes four parameters: 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 the parser) to the ASTs for the `ident` and `expr` that it has already 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 parser assigns its result to `RESULT`, which is a special symbol that represents the left-hand side non-terminal (`stmt`). As it matches productions, the parser builds and merges AST subtrees into progressively larger trees. Finally, when it matches the first production (corresponding to an entire program), the parser returns the root of the complete AST. The bookkeeping required to maintain AST pointers as the parser matches productions is automatic.

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 by `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. are all instances of class `CSXToken`, whereas the `IDENTIFIER` token is an instance of class `CSXIdentifierToken`. The lines

```
non terminal csxLiteNode       prog;
non terminal StmtsNode         stmts;
```

say that the nonterminal `prog` references class `csxLiteNode`, whereas the nonterminal `stmts` references `StmtsNode`.

The function `parse()` returns an object of type `Symbol`. For successful parses, it is the start symbol (`program`) of the derivation. The `value` field of the returned `Symbol` contains 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 so 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. The line numbers printed *might 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) get a line number that appears "out of order" because the line number stored with an AST node corresponds to where the construct *begins*.

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 we always print on a new line, we 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 `IntLitNodes`, we print `intval`. For `StrLitNodes`, we print `strval` (the full string representation, with quotes and escapes). For `CharLitNodes`, print `charval` as a quoted character in fully escaped form. For `IdentNodes`, the unparser uses `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, add explicit parentheses to guarantee that expressions are properly interpreted. Hence `A+B*C` should 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 gets extra credit.)

What You Must Do

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

program	→	{ stmts }
stmts	→	stmt stmts
	|	λ
stmt	→	id = expr ;
	|	if (expr) stmt
expr	→	expr + id
	|	expr - id
	|	id

CSX_lite Grammar

This simple subset contains no declarations, only an assignment and **if** statement, and expressions involving only +, **-** and identifiers. Complete specifications, parsers, AST builders and unparsers for CSX_lite may be found in ~raphael/-courses/cs541/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. It builds ASTs using calls to constructors as illustrated in csx_lite.cup. Once the parser matches an individual production, it calls a constructor for the corresponding AST node. You should substitute your scanner from Project 2, by replacing csx_lite.jlex with your csx_go.jlex file.

Unparsing functions, one for each type of AST node, are member functions in ast.java. Each such routine is fairly simple — it prints the information in the node 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 **for** loop. First, add the appropriate productions to the *JavaCUP* 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 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 verify that the ASTs you build 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_go productions to your CUP specification, eventually creating a complete CSX_go parser and unparser.

Error Handling

In the case of syntax errors, *JavaCUP* calls syntax_error() to print an error message and then throws a SyntaxErrorException, indicating abnormal termination. The caller of your parser should catch this exception, which indicates that because of errors it cannot build an AST.

JavaCUP does provide a simple error recovery mechanism (using "error" markers). This feature is described in §5 of the CUP manual. If you wish, you may experiment with syntactic error recovery *after* your parser is fully operational. It is not necessary to continue parsing after parse() discovers a syntax error.

What to Hand In

As input, your parser takes a text-file name on the command line, which it passes 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, line-numbered, unparsed listing of the resulting abstract syntax tree. Turn in a your parser module, your CUP specification, and a listing of your parser's execution on a variety of syntactically valid and invalid programs.

If you wish to claim extra credit, *clearly* state (in the README file) what you've added, and include examples of its operation. In particular, if you implement labeled **for** statements, **break** and **continue**, mention that fact.