CS 536 — Spring 1997

Programming Assignment #3

CSX Parser

Due: Tuesday, April 1, 1997
Not accepted after class on Tuesday, April 8, 1997

You are to write a C++ parser module and a Yacc parser specification to implement a parser for CSX. A grammar that defines CSX's syntax appears on the next page. You should examine the grammar carefully to learn the structure of CSX constructs. In most cases, structures are very similar to those of C 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 ! 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 (i.e., `A==B==C` is illegal). The unary ! operator is (of course) right-associative. Be sure that your parser for CSX properly reflects these precedence and associativity rules.

Using Yacc to Build a Parser

You will use Yacc, the Unix parser generator, to build your CSX parser. You'll have to rewrite the CSX grammar into the format required by Yacc. This format is defined in §6.7.2 as well as the Yacc manual ("Yacc: Yet Another Compiler-Compiler"). A sample Yacc specification corresponding to CSX-lite (a small subset of CSX) may be found in ~cs536-1/public/proj3/c++/lite1.yacc.

Once you've rewritten the CSX grammar we've provided and entered it into a file (say csx.yacc), you can test whether the grammar can be parsed by a Yacc-generated parser. Run

```
yacc csx.yacc
```

Yacc should generate the message

```
conflicts: 4 shift/reduce
```

This indicates that the grammar we've provided is almost, but not quite, in a form acceptable to Yacc. This is a common occurrence. Most context-free grammars used to define programming languages can be handled by Yacc, sometimes after minor modification.

If you examine the parsing context immediately after the token main is matched, you'll see there is a problem when the next input token is int (or bool). In particular, Yacc can't decide whether to match the int as part of a variable declaration (perhaps using "decl → type id ;") or to let the non-terminal decls derive λ and match int as part of a procedure declaration.

```
program    →  main id ; decls procdecls stmts end
decls      →  decl decls
           |  λ
procdecls  →  procdecl procdecls
           |  λ
decl       →  type id ;
           |  type id [ intlit ] ;
           |  type id = exp ;
           |  const id = exp ;
procdecl   →  proc id ( ) ; decls stmts end ;
           |  proc id ( argdecls ) ; decls stmts end ;
           |  type proc id ( ) ; decls stmts end ;
           |  type proc id ( argdecls ) ; decls stmts end ;
type       →  int
           |  bool
argdecls   →  argdecl , argdecls
           |  argdecl
argdecl    →  type id
           |  type id &
           |  type id [ ] &
stmts      →  stmt stmts
           |  stmt
stmt       →  name = exp ;
           |  if exp then stmts end ;
           |  if exp then stmts else stmts end ;
           |  while exp loop stmts end ;
           |  readlist ;
           |  writelist ;
           |  id ( ) ;
           |  id ( args ) ;
           |  return exp ;
           |  return ;
name       →  id
           |  id [ exp ]
args       →  exp , args
           |  exp
readlist   →  >> name readlist
           |  >> name
writelist  →  << item writelist
           |  << item
item       →  strlit
           |  exp
exp        →  exp || term
           |  exp && 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
        |   unit
unit    →   name
        |   id ( )
        |   id ( args )
        |   intlit
        |   true
        |   false
        |   ( exp )
```

CSX Grammar

You'll have to rewrite that part of the grammar to make it easier for the parser to make the decision of how to match int. You **can't** change the CSX language itself (i.e., the sequences of tokens considered valid), but to *can* change the productions used to generate declarations of variables and procedures after main. You may find it helpful to consider the techniques used to make grammars LL(1) (see § 5.6). Though Yacc accepts a richer class of grammars than LL(1), Yacc accepts (and can parse) *every* LL(1) grammar.

Yacc allows expressions to be defined in a more compact and readable form. Production rules of the form

```
exp : exp op exp ;
```

are allowed, even though such productions are ambiguous. The precedence and associativity of the corresponding operators (e.g., op) are defined in Yacc's declaration section. This process is detailed in §6.7.3 of the text and §6 of the Yacc manual. You may use productions of this form to simplify your CSX grammar if you wish.

Once your grammar is in the right format and generates no "conflict" messages, enter

```
yacc -d csx.yacc
```

Yacc will create a file y.tab.c that contains the parser it has generated. It will also create a file y.tab.h that contains the token codes the parser is expecting. Use y.tab.h with flex in generating your scanner to guarantee that both the scanner and parser use the same token codes.

The generated parser is named yyparse(). It expects to call a scanner created by Lex or Flex. When you call yyparse(), it will repeatedly call the scanner to parse the input program. It will return 0 if the parse is successful. If there is a syntax error it will return 1. It will also call a routine yyerror(char *) to print an error message. We provide a simple implementation of yyerror in the code skeletons. 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 C++ 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:

```
class ASTNode { // abstract superclass; only subclasses are actually created

  public:
        const  int    linenum;
        const  int    colnum;
        ASTNode(int l, int c) : linenum(l), colnum(c){}; // set linenum & colnum values
        virtual void Unparse(int indent = 0) = 0;
};
```

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 constant fields: linenum and colnum. These 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 a constructor that sets linenum and colnum. This constructor is called by constructors of subclasses to set these two fields. The virtual function 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. The function is virtual because we require that each subclass provide its own definition of Unparse. 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.

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

```
class programNode : public ASTNode {
  public:
    programNode(identNode*, declsNode*, procDeclsNode*, stmtsNode*, int line, int col);
    void Unparse(int indent = 0);
  private:
    identNode       *progName;
    declsNode       *progDecls;
    procDeclsNode   *progProcs;
    stmtsNode       *progStmts;
};
```

programNode corresponds to the start symbol of all CSX programs, program. programNode 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. Since Unparse is virtual, this class, along will all other ASTnode classes, will provide a definition of Unparse appropriate to the particular CSX program structure the class represents.

programNode also contains four private fields. These correspond to the four subtrees a programNode will contain. These correspond in turn to the program's name (an identifier), the program's global variable and constant declarations, the program's procedure declarations, and the program's body (executable statements). 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 progName, we'd get a C++ type error, because the AST node type corresponding to integer literals (intLitNode) is different that the type progName 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 now list all the AST nodes (and their classes) that we use. This information appears below. For each class used to represent AST nodes, we'll list the private field names in that class and the type of each field. This type will usually be a pointer to a particular AST class. In some cases a field may point to a special AST node, the nullNode. The nullNode is an AST node that corresponds to λ. That is, if a subtree is empty, we'll use a nullNode to represent that fact. For example, in a program, procedure declarations are optional. If a particular program chooses to have no procedures, the progProcs field in its programNode will point to a nullNode. As you might expect, a nullNode has no internal fields. It simply serves as a placeholder so that all subtrees that are expected are always present.

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 and strLitNode do not have subtrees, but do contain the serial number, integer value, or string values returned by the scanner (in yylval). Leaf nodes like trueNode and boolTypeNode use 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.

A nullNode is used to represent null subtrees. It is better to point to a nullNode than to store a null pointer. If all pointers in AST nodes point to *something* then we never have to check a pointer 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. Then why do we bother to define it?

Sometimes we want to be able to point to one of a number of kinds of AST nodes, but not to just any node. Thus in a stmtsNode we want to point to any kind of AST node corresponding to a statement, but not to AST nodes corresponding to non-statements. We solve this problem by pointing to a node of type stmtNode. We make all classes corresponding to statements (like asgNode or readNode) *subclasses* of stmtNode. The rules of C++ say that a variable declared to be a pointer to a class S may be assigned a pointer to any *subclass* of S. This is because a subclass of S contains everything S does (and perhaps more). Thus a pointer to an asgNode may be assigned to a variable expecting a pointer to a stmtNode without error. However a pointer to an AST node that is not a subclass of stmtNode (e.g., boolTypeNode) may not be legally assigned to a variable expecting a stmtNode pointer.

Although the set of class definitions in ast.h 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 your are trying to assign to an AST node's field is illegal. In the table below we list all the AST nodes that appears in ast.h and their superclass. Note that in all cases, the superclass is abstract.

class	Fields Used	Type of Fields	nullNode allowed?
programNode	progName progDecls progProcs progStamts	identNode∗ declsNode∗ procDeclsNode∗ stmtsNode∗	No Yes Yes No
declsNode	thisDecl moreDecls	declNode∗ declsNode∗	No Yes
procDeclsNode	thisDecl moreDecls	procDeclNode∗ procDeclsNode∗	No Yes
varDeclNode	varName varType initValue	identNode∗ typeNode∗ exprNode∗	No No Yes
constDeclNode	constName constValue	identNode∗ exprNode∗	No No
arrayDeclNode	arrayName elementType arraySize	identNode∗ typeNode∗ intLitNode∗	No No No
procDeclNode	procName procArgs procReturnType procDecls procStmts	identNode∗ argDeclsNode∗ typeNode∗ declsNode∗ stmtsNode∗	No Yes Yes Yes No
argDeclsNode	thisDecl moreDecls	argDeclNode∗ argDeclsNode∗	No Yes
refArgDeclNode	argName argType	identNode∗ typeNode∗	No No
refArrayDeclNode	argName elementType	identNode∗ typeNode∗	No No
valArgDecl	argName argType	identNode∗ typeNode∗	No No
intTypeNode			
boolTypeNode			
stmtsNode	thisStmt moreStmts	stmtNode∗ stmtsNode∗	No Yes
asgNode	target source	nameNode∗ exprNode∗	No No
ifThenNode	condition thenPart elsePart	exprNode∗ stmtsNode∗ stmtsNode∗	No No Yes
whileLoopNode	condition loopBody	exprNode∗ stmtsNode∗	No No
readNode	targetVar moreReads	nameNode∗ readNode∗	No Yes

Fields Used in astNode (Part 1)

class	Fields Used	Type of Fields	nullNode allowed?
writeNode	outputValue moreWrites	writeValueNode* writeNode*	No Yes
callNode	procName procArgs	identNode* argsNode*	No Yes
returnNode	returnVal	exprNode*	Yes
nameNode	varName subscriptVal	identNode* exprNode*	No Yes
argsNode	argVal moreArgs	exprNode* argsNode*	No Yes
binaryOpNode	leftOperand rightOperand operatorCode	exprNode* exprNode* int	No No No
unaryOpNode	operand operatorCode	exprNode* int	No No
identNode	idval	serialNum	No
fctCallNode	functionName functionArgs	identNode* argsNode*	No Yes
intLitNode	intval	int	No
strLitNode	strval fullstr	char* char*	No No
trueNode	none		
falseNode	none		
nullNode	none		

Fields Used in astNode (Part 2)

AST Node	Superclass	AST Node	Superclass
argDeclNode	ASTNode	argDeclsNode	ASTNode
argsNode	ASTNode	arrayDeclNode	declNode
asgNode	stmtNode	binaryOpNode	exprNode
boolTypeNode	typeNode	callNode	stmtNode
constDeclNode	declNode	declNode	ASTNode
declsNode	ASTNode	exprNode	writeValueNode
falseNode	exprNode	fctCallNode	exprNode
identNode	exprNode	ifThenNode	stmtNode
intLitNode	exprNode	intTypeNode	typeNode
nameNode	exprNode	nullNode	ASTNode
procDeclNode	ASTNode	procDeclsNode	ASTNode
programNode	ASTNode	readNode	stmtNode
refArgDeclNode	argDeclNode	refArrayDeclNode	argDeclNode
returnNode	stmtNode	stmtNode	ASTNode
stmtsNode	ASTNode	strLitNode	writeValueNode
trueNode	exprNode	typeNode	ASTNode
unaryOpNode	exprNode	valArgDeclNode	argDeclNode
varDeclNode	declNode	whileNode	stmtNode
writeNode	stmtNode	writeValueNode	ASTNode

Classes Used in AST Nodes and Their Superclasses

Getting Started

One of the first things you'll need to do is to create constructors for all the AST node classes we've defined. This is easy to do. As usual, we've placed skeleton files for the project in ~cs536-1/public/proj3/c++. Look at file ast.cc. We have included definitions for a number of AST nodes used in **CSX-lite**, a small subset of CSX. You need to complete the remaining constructors. Essentially all you need to do is assign parameters from the constructor to the appropriate fields of the class. Initialization of linenum and colnum (defined in ASTNode) is already taken care of by calls to constructors of the appropriate superclass.

Building ASTs

Once we have constructors for all the AST classes, we need to use them to build ASTs for CSX programs we have parsed. One of the reasons we're using Yacc to build our parser is that it's easy to build ASTs using Yacc. Yacc allows us to embed *actions*, in the form of C++ code, in the productions Yacc parses. When a production containing an action is matched by yyparse(), the associated action is automatically executed. For example in the following rule (drawn from lite.yacc)

```
stmt  : ident '=' exp ';'
                {$$=new asgNode($1,$3,$1->linenum,$1->colnum);}
```

the production stmt $\rightarrow$ ident = expr ; is specified. Moreover, whenever this production is matched, the constructor asgNode is called (since asgNode correspond to assignment statements). The constructor for asgNode wants four things: pointers to ASTs corresponding to the source and target of the assignment, and a line and column number to associate with the assignment. The special symbols $1 and $3 represent pointers (automatically maintained by Yacc) to the ASTs for the ident and expr that have already been parsed, (these are the first and third symbols on the production's righthand side). They have already been built by the time this production is matched. We'll 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, $1 points to the AST node built for ident, $1->linenum represent 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 $$. $$ is a special symbol, maintained by Yacc, that represents the AST for 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 Yacc, using the $$, $1, $2, ... notation.

Information placed in yylval by the scanner can also be easily accessed. If the n-th symbol in a righthand side is a terminal, then $n denotes the value of yylval when that terminal was matched. Thus the rule

```
exp  : exp '+' exp
       {$$=new binaryOpNode($1,'+',$3,$2.linenum,$2.colnum);}
```

uses the linenum and colnum values of the '+' (extracted as $2.linenum and $2.colnum) in constructing a binaryOpNode that represents the AST for the addition operation.

The data that may appear in a Yacc symbol (referenced by $$ or $n) is defined in a %union declaration. Thus

```
%union {
                Token           t;
                programNode     *prog;
                stmtsNode       *stmts;
                stmtNode        *stmt;
                nameNode        *name;
                exprNode        *expr;
}
```

tells Yacc that during parsing terminal and nonterminal symbols may contain Token structs, or programNode pointers, or stmtsNode pointers, etc. The %type declaration tells Yacc the field name to use for particular non-terminals. Thus

```
%type <prog> prog
%type <stmts> stmts
%type <stmt> stmt
%type <expr> exp
%type <name> ident
```

says that all nonterminals prog will extract their values during parsing using the field prog (and hence will point to a programNode). Similarly, all occurrences of a stmt non-terminal will use the field stmts and hence will point to a stmtNode. For terminals, the field name to be used is included in the %token declaration. For example,

```
%token  <t>  ID
```

says that IDs will have a Token struct associated with them (supplied by the scanner in yylval).

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 end of the main program) 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() (to print the target variable, without indenting), print " = ", call source->Unparse() (to print the source expression, without indenting), and finally print ";".

For intLitNodes we print intval. For strLitNodes we print fullstr (using the original string representation, with quotes and escapes). For identNode the unparser should use symToStr to obtain 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 Yacc to help you build your parser. Still, it helps a lot to see an example of all the pieces you'll need to complete in your project. We've created a small subset of CSX, called CSX-lite, that's defined by the following productions:

program	→	main stmts end
stmts	→	stmt stmts
	|	λ
stmt	→	id = exp ;
	|	if exp then stmts end ;
exp	→	exp + id
	|	exp – 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 Yacc specifications, parsers, AST builders and unparsers for CSX-lite may be found in ~cs536-1/public/proj3/c++. 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.yacc. Once an individual production is matched by the parser, a constructor for the corresponding AST node is called.

Unparsing functions, one for each AST node that is actually built, are in unparse.cc. 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 Yacc 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 Yacc 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 Yacc specification, eventually creating a complete CSX parser and unparser.

Error Handling

In the case of syntax errors Yacc will call yyerror() to print a syntax error message and then return a 1, indicating abnormal termination. Your parser should return a null abstract syntax tree pointer, indicating that no AST could be built (because the input was invalid).

Yacc does provide a simple error recovery mechanism (using "error" markers). This is described in §17.2.9 of the text and §7 of the Yacc 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. Your should hand in a listing from Yacc to show your CSX grammar and verify that it actually is valid. 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 Yacc specification and a listing of your parser's execution on a variety of syntactically valid and invalid programs. Also, place an executable copy of your program in ~cs536-1/public/proj3/handin. The name of the program should be your login (e.g., I would create ~cs536-1/public/proj3/handin/fischer). Your program should take the name of a CSX source file on the command line, just as all Unix compilers do. We will of course run your program on test cases of our own.

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