Computer Science 330 Language Implementation Test

6.20-8.00pm Thursday 6th April 2006

Course Compsci 330/Compsci 601
Surname
Given Names
Student ID Number
Login Name
Normal Signature

1 ______________________________ /25

2(a) ___________________________ /20

2(b) ____________________________ /5

2(c)_____________________________ /20

2(d) ____________________________ /5

2(e)_____________________________ /5

3 ______________________________ /20

Total _________________________ /100

Start reading 6.20p.m. Write your name on all sheets of your answer book. Start writing your answers at 6.30pm. Stop writing at 8.00p.m.

Remove the staple fastening the appendices to the answer book, but do not remove the staples from the answer book. Read the questions carefully. Hand in your answer book at the front of the class. Attempt all questions. Questions total 100 marks. The test counts for 20% of the total mark.

Question 1 25 Marks

Write JFlex regular expressions to match the following tokens. You may declare support regular expressions if you need them.

(a) An identifier, composed of an upper case letter, followed by 0 or more letters or digits. For example, `Happy2`, but not `sad1`.

(2 marks)

(b) An identifier, composed of a sequence of one or more letters, followed by 0 or more digits. For example, `base`, `base16`, but not `base16Number`.

(2 marks)

(c) An identifier, composed of a single alphabetic letter, followed by an optional single digit. For example, `I`, `U2`, but not `Eye`, `You2`, `U24`, `UB40`.

(2 marks)

(d) A hexadecimal integer, with a prefix of `0x` or `0X`, where all digits have to agree with the case of the prefix. For example, `0x123456789abcdef`, or `0X123456789ABCDEF`, but not `0x123456789AbCdEf`, or `0x123456789ABCDEF`.

(2 marks)

(e) An identifier, composed of one or more words, with "_" characters in between, where a word is an upper case letter, followed by 0 or more lower case letters. For example, `The_Cat_Is_Hungry`, but not `The_Cat_Is_`, or `The_cat`, or `TheCat_IsHungry`.

(6 marks)

Print your login name____

(f) A decimal integer, between `0` and `255`, without unnecessary leading "`0`"s. For example, `0`, `1`, `98`, `128`, `254`, `255`, but not `01`, `256`, `330`.

(6 marks)

(g) Indicate five different kinds of errors in the following fragment of JFlex code. ("..." just means omitted code). Assume spaces and line breaks are not syntactically important.

```
...
newline     =     \r|\n|\r\n
space       =     [\ \t]
ident       =     ...
%state NORMAL, COMMENT
%%
<NORMAL> {
      {ident}      { return token( sym.IDENT ); }
      if           { return token( sym.IF ); }
      else         { return token( sym.ELSE ); }
      ...
      /*           { yybegin( NORMAL ); }
      {space}      { return token( sym.SPACE ); }
      {newline}    { linecount++; }
      .            { }
      }
<COMMENT> {
      */           { yybegin( NORMAL ); }
      .            { }
      {newline}    { linecount++; }
      }
```

1

2

3

4

5

(5 marks)

Question 2 **55 marks**

Consider the CUP grammar in the appendices. The terminal symbols LEFTBRACE, RIGHTBRACE, ASSIGN, SEMICOLON, COMMA correspond to "{", "}", "=", ";", ",", and IDENT corresponds to an identifier.

(a) Using the information provided in the appendices, perform a shift-reduce LALR(1) parse of the valid input

```
int a = { b, c }, d;
```

Show both the symbols and states on the stack, the current token, and the action performed at each stage. (Consider "int" to be an IDENT).

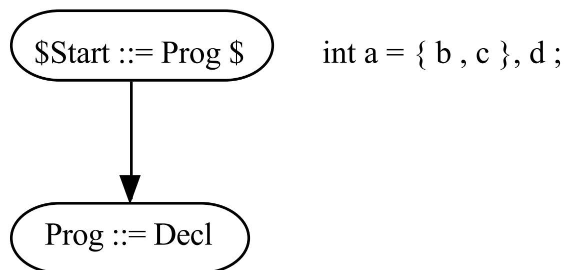
Note that some grammar rules are left recursive, while others are right recursive. (20 marks)

Stack								Input	Action	
$0								ID int	Shift	ID 4
$0	ID 4							ID a	Reduce	Type ::= ID
$0	Type 2								Shift	ID 8
$0	Type 2	ID 8						=	Shift	= 14
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0										
$0	Decl 5								Reduce	Prog ::= Decl
$0	Prog 1								Shift	$ 25
$0	Prog 1	$ 25						$	Reduce	$Start ::= Prog $
$0	$Start -1								Accept	

Print your login name____

(b) Draw the full parse tree, showing all rules used in the above shift-reduce LALR(1) parse.

(5 marks)



(c) Using the information provided in the appendices, perform a shift-reduce LALR(1) parse of the invalid input

```
int a { b, c, d };
```

giving a clear indication of how CUP processes the input in the event of an error.

Assume error_sync_size() returns 1, so you do not need to first parse and check three tokens can be consumed without generating an error, before reparsing and performing the actions. (20 marks)

Stack								
$0								
$0	ID 4							
$0	Type 2							
$0	Type 2	ID 8						
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0								
$0	Decl 5							
$0	Prog 1							
$0	Prog 1	$ 25						
$0	$Start -1							

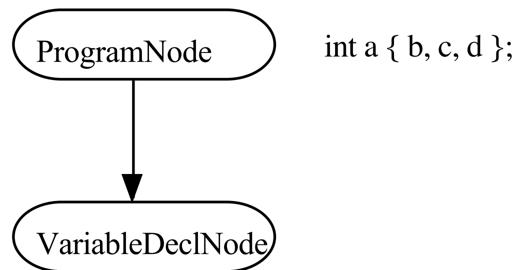
Input	Action	
ID int	Shift	ID 4
ID a	Reduce	Type ::= ID
	Shift	ID 8
{	Error	
	Reduce	Prog ::= Decl
	Shift	$ 25
$	Reduce	$Start ::= Prog $
	Accept	

Print your login name____

(d) Draw the abstract syntax tree, as specified by the actions associated with the rules.

Assume DeclrListNode is composed of a vector, containing all the children.

(5 marks)



(e) Write Java code to implement an ExprListNode, with a toString() method to reprint the list, taking into account that the tail might be null.

Assume ExprListNode is used to create a right recursive linked list, with the same structure as the grammar rule.

(5 marks)

```
package node.exprNode;

import node.*;

public class ExprListNode extends [        ] {

[                                                    ]

    public ExprListNode(
[                                                    ]
        ) {
[                                                    ]
        }

    public String toString() {
[                                                    ]
        }
    }
```

Print your login name___________________________________

Question 3 20 marks

Suppose we have a language for writing down expressions involving integer constants, simple variables and function invocations (in a C or Java style, with 0 or more comma separated actual parameters, enclosed in parentheses, even if there are 0 parameters). For example, we can write expressions of the form:

```
a( 3, b(), c, d( 4, 5 ), e( f, g( 6 ) ) )
```

Moreover, we can follow the expression by an optional "where declaration list", which defines the meaning of some of the variables and functions, in a comma separated sequence of declarations. For example, we may define:

```
a( 3, b(), c, d( 4, 5 ), e( f, g( 6 ) ) )
      where
            b() = 4,
            c = 7,
            d( x, y ) = x,
            e( x, y ) = d( y, x ),
            g( x ) = h( x, 1, d( x, x ) )
```

The left hand side of a declaration represents the variable or function being defined, together with its formal parameters, in the case of a function, and the right hand side represents its value, in terms of other variables and functions, and the formal parameters of the function being declared. Note that the variables and formal parameters are identifiers, not expressions.

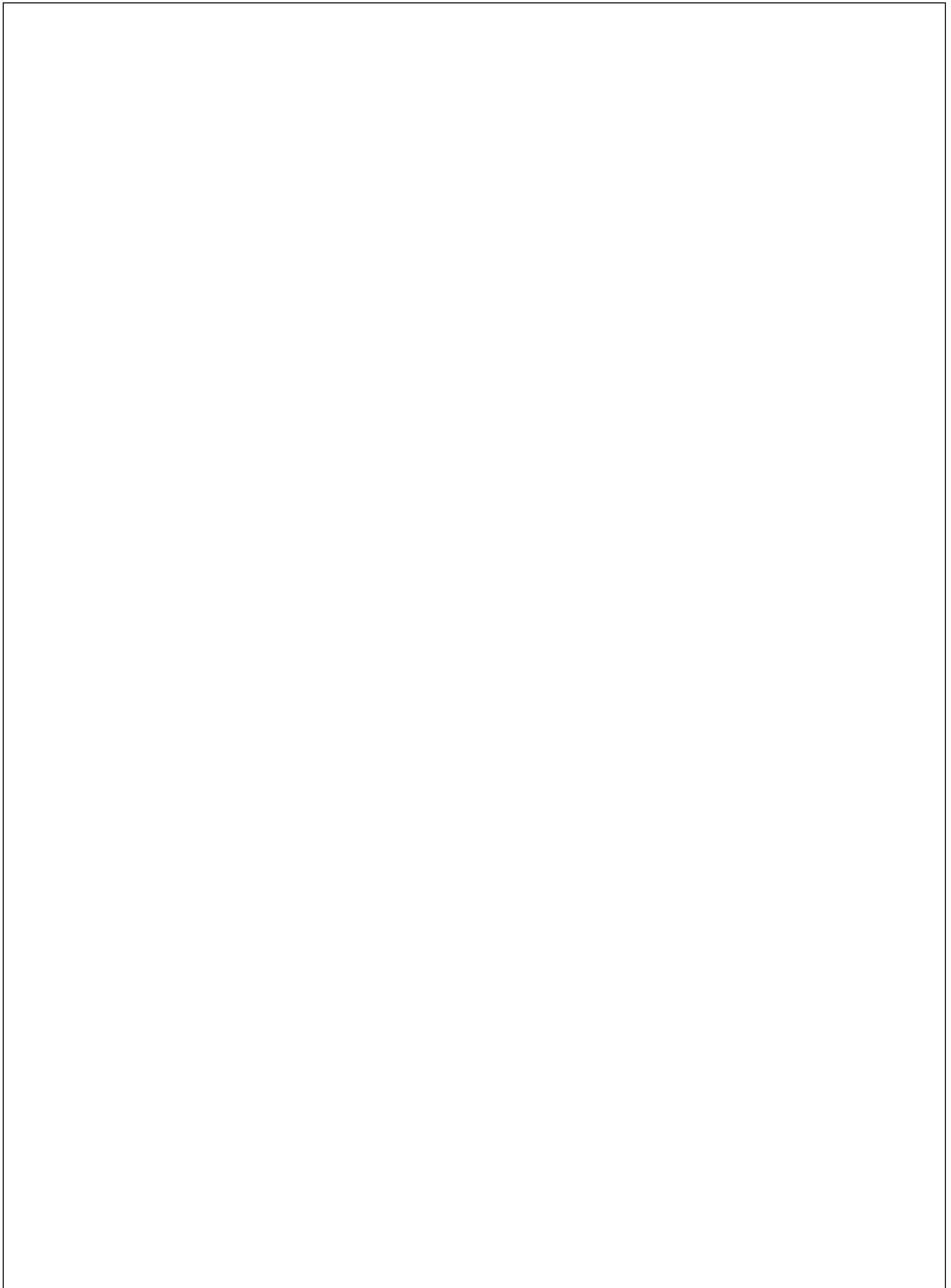
Presumably a "compiler" will "simplify" the above "program" into:

```
a( 3, 4, 7, 4, h( 6, 1, 6 ) )
```

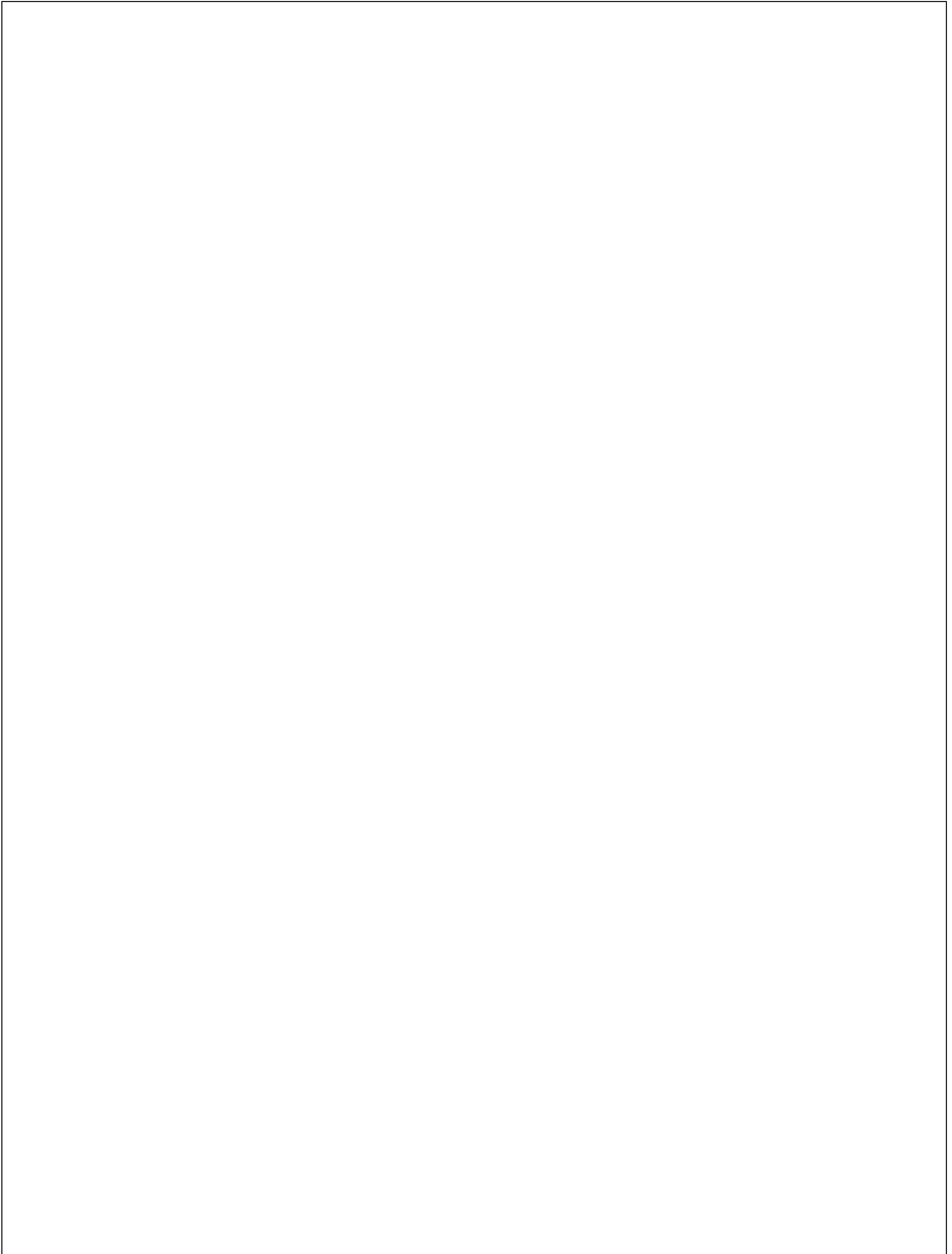
by performing as many substitutions as possible.

Write a CUP grammar definition to parse input for the above language. You do not have to write any actions.

(20 Marks)



Print your login name____



Appendices

The CUP grammar

```
terminal ERROR, LEFTBRACE, RIGHTBRACE, ASSIGN, SEMICOLON, COMMA;
terminal String IDENT;

nonterminal ProgramNode Program;
nonterminal DeclNode Decl;
nonterminal TypeNode Type;
nonterminal DeclrListNode DeclrList;
nonterminal DeclrNode Declr;
nonterminal ExprNode Expr;
nonterminal ExprListNode ExprList;

start with Program;

Program::=
            Decl:decl
            {:
            RESULT = new ProgramNode( decl );
            :}
      ;

Decl::=
            Type:type DeclrList:declrList SEMICOLON
            {:
            RESULT = new VariableDeclNode( type, declrList );
            :}
      |
            error SEMICOLON
            {:
            RESULT = new ErrorDeclNode( "DeclError...;" );
            :}
      ;

Type::=
            IDENT:ident
            {:
            RESULT = new TypeIdentNode( ident );
            :}
      ;

DeclrList::=
            Declr:declr
            {:
            RESULT = new DeclrListNode( declr );
            :}
      |
            DeclrList:declrList COMMA Declr:declr
            {:
            declrList.addElement( declr );
            RESULT = declrList;
            :}
      ;
```

```
Declr::=
            IDENT:ident ASSIGN Expr:expr
            {:
            RESULT = new InitDeclrNode( ident, expr );
            :}
      |
            IDENT:ident
            {:
            RESULT = new UninitDeclrNode( ident );
            :}
      |
            error
            {:
            RESULT = new ErrorDeclrNode( "DeclrError..." );
            :}
      ;

Expr::=
            LEFTBRACE ExprList:exprList RIGHTBRACE
            {:
            RESULT = new CompoundExprNode( exprList );
            :}
      |
            LEFTBRACE error RIGHTBRACE
            {:
            RESULT = new ErrorExprNode( "{ ExprListError ... }" );
            :}
      |
            IDENT:ident
            {:
            RESULT = new VariableExprNode( ident );
            :}
      ;

ExprList::=
            Expr:expr
            {:
            RESULT = new ExprListNode( expr, null );
            :}
      |
            Expr:expr COMMA ExprList:exprList
            {:
            RESULT = new ExprListNode( expr, exprList );
            :}
      ;Tables for the CUP grammar
```

Grammar Rules (Productions)

```
[0] $START ::= Program EOF
[1] Program ::= Decl
[2] Decl ::= Type DeclrList SEMICOLON
[3] Decl ::= error SEMICOLON
[4] Type ::= IDENT
[5] DeclrList ::= Declr
[6] DeclrList ::= DeclrList COMMA Declr
[7] Declr ::= IDENT ASSIGN Expr
[8] Declr ::= IDENT
[9] Declr ::= error
[10] Expr ::= LEFTBRACE ExprList RIGHTBRACE
[11] Expr ::= LEFTBRACE error RIGHTBRACE
[12] Expr ::= IDENT
[13] ExprList ::= Expr
[14] ExprList ::= Expr COMMA ExprList
```

Action Table

```
From state #0
     error:SHIFT(state 3) IDENT:SHIFT(state 4)
From state #1
     EOF:SHIFT(state 25)
From state #2
     error:SHIFT(state 7) IDENT:SHIFT(state 8)
From state #3
     SEMICOLON:SHIFT(state 6)
From state #4
     error:REDUCE(rule 4) IDENT:REDUCE(rule 4)
From state #5
     EOF:REDUCE(rule 1)
From state #6
     EOF:REDUCE(rule 3)
From state #7
     SEMICOLON:REDUCE(rule 9) COMMA:REDUCE(rule 9)
From state #8
     ASSIGN:SHIFT(state 14) SEMICOLON:REDUCE(rule 8) COMMA:REDUCE(rule 8)
From state #9
     SEMICOLON:SHIFT(state 12) COMMA:SHIFT(state 11)
From state #10
     SEMICOLON:REDUCE(rule 5) COMMA:REDUCE(rule 5)
From state #11
     error:SHIFT(state 7) IDENT:SHIFT(state 8)
From state #12
     EOF:REDUCE(rule 2)
From state #13
     SEMICOLON:REDUCE(rule 6) COMMA:REDUCE(rule 6)
From state #14
     LEFTBRACE:SHIFT(state 15) IDENT:SHIFT(state 17)
From state #15
     error:SHIFT(state 19) LEFTBRACE:SHIFT(state 15) IDENT:SHIFT(state 17)
From state #16
     SEMICOLON:REDUCE(rule 7) COMMA:REDUCE(rule 7)
From state #17
     RIGHTBRACE:REDUCE(rule 12) SEMICOLON:REDUCE(rule 12) COMMA:REDUCE(rule 12)
From state #18
     RIGHTBRACE:SHIFT(state 24)
From state #19
     RIGHTBRACE:SHIFT(state 23)
From state #20
     RIGHTBRACE:REDUCE(rule 13) COMMA:SHIFT(state 21)
From state #21
     LEFTBRACE:SHIFT(state 15) IDENT:SHIFT(state 17)
From state #22
     RIGHTBRACE:REDUCE(rule 14)
From state #23
     RIGHTBRACE:REDUCE(rule 11) SEMICOLON:REDUCE(rule 11) COMMA:REDUCE(rule 11)
From state #24
     RIGHTBRACE:REDUCE(rule 10) SEMICOLON:REDUCE(rule 10) COMMA:REDUCE(rule 10)
From state #25
     EOF:REDUCE(rule 0)
```

Reduce (GoTo) Table

```
From state #0:
     Program:GOTO(1)
     Decl:GOTO(5)
     Type:GOTO(2)
From state #1:
From state #2:
     DeclrList:GOTO(9)
     Declr:GOTO(10)
From state #3:
From state #4:
From state #5:
From state #6:
From state #7:
From state #8:
From state #9:
From state #10:
From state #11:
     Declr:GOTO(13)
From state #12:
From state #13:
From state #14:
     Expr:GOTO(16)
From state #15:
     Expr:GOTO(20)
     ExprList:GOTO(18)
From state #16:
From state #17:
From state #18:
From state #19:
From state #20:
From state #21:
     Expr:GOTO(20)
     ExprList:GOTO(22)
From state #22:
From state #23:
From state #24:
From state #25:
```