1. LR Parsing

Given the following grammar:

\[ S \rightarrow \varepsilon A \]
\[ A \rightarrow AmB \mid a \]
\[ B \rightarrow bB \mid a \]

(1) [15p] Construct the LR parsing finite state machine, including the states and transitions;
(2) [10p] Fill in the following parsing table (more rows are needed than depicted)

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>m</th>
<th>$</th>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>$1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$2</td>
<td></td>
<td></td>
<td></td>
<td>$3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$3</td>
<td>$3</td>
<td>$3</td>
<td>$3</td>
<td>$3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>$4</td>
<td></td>
<td>$6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$5</td>
<td>$7</td>
<td></td>
<td></td>
<td>$8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$5</td>
<td>$5</td>
<td>$5</td>
<td>$5</td>
<td>$5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$5</td>
<td>$7</td>
<td></td>
<td></td>
<td>$8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$4</td>
<td>$4</td>
<td>$4</td>
<td>$4</td>
<td>$4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grammar Rules: ($\Gamma_k$ means reduce using rule $k$), $AC=$Accept.

1. $S \rightarrow \ast A$
2. $A \rightarrow AmB$
3. $A \rightarrow a$
4. $B \rightarrow bB$
5. $B \rightarrow a$. 
(3) Parse the input string *ama by filling the following parsing table. The number 1 in the stack column indicates the first state is 0.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*ama$</td>
<td>[0, $] = \text{S1.}</td>
</tr>
<tr>
<td>0A</td>
<td>ama$</td>
<td>[1, a] = \text{S2}</td>
</tr>
<tr>
<td>0A²</td>
<td>ma$</td>
<td>[2, m] = \text{Reduce: A} \Rightarrow a</td>
</tr>
<tr>
<td>0A³</td>
<td>ma$</td>
<td>[1, A] = q³</td>
</tr>
<tr>
<td>0A⁴</td>
<td>ma$</td>
<td>[3, m] = \text{S4}</td>
</tr>
<tr>
<td>0A⁵</td>
<td>a$</td>
<td>[4, a] = \text{S5}</td>
</tr>
<tr>
<td>0A⁶</td>
<td>$</td>
<td>[5, $] = \text{Reduce: B} \Rightarrow a</td>
</tr>
<tr>
<td>0A⁷</td>
<td>$</td>
<td>[4, B] = q⁶</td>
</tr>
<tr>
<td>0A⁸</td>
<td>$</td>
<td>[6, $] = \text{Reduce: A} \Rightarrow \text{AmB}</td>
</tr>
<tr>
<td>0A⁹</td>
<td>$</td>
<td>[1, A] = q³</td>
</tr>
<tr>
<td>0A¹⁰</td>
<td>$</td>
<td>[3, $] = \text{ACCEPT}</td>
</tr>
</tbody>
</table>

\[ S \Rightarrow \text{X A} $ \]
2. Visitor pattern

(1) [10p] Please explain the benefits of the visitor pattern.

Visitor pattern separates the data storage and algorithm, providing a more flexible way to process/traverse the data. Just do a little hand shake, using the class polymorphism, it will be easier to maintain the code.

(2) [15p] Please fill in the following visitor-pattern-related code snippet to regenerate code from a given AST. For example, the visitor pattern should be able to print out the below program (or its equivalence) by visiting the given AST.

```
ifStmt
  |____________________
  |____________________
GreaterThanExpr
  |____________________
  |____________________
IdExpr	NumExpr
  |____________
    x	2
AssignStmt
  |____________________
  |____________________
IdExpr	MulExpr
  |____________
    y	NumExpr
      |____________
        AddExpr
          |___________
            IdExpr	NumExpr
              |____________
                IdExpr	NumExpr
                  |___________
                    AssignStmt
                      |___________
                        AssignStmt
                          |___________
                            AssignStmt
```

if (x>2) {
    y=3*(x+1);
} else {
    x=0;
    y=0;
}

Complete the body of the empty methods in following code snippet (feel free to change the return type of the accept() and visit() methods if needed):
Using print / println to express

public class AddExpr extends Expr {
    Expr e1, e2;
    public void accept (Visitor v) {
        V. visit (this);
    }
}

public class IfStmt extends Stmt {
    Expr e;
    Stmt s1, s2;
    public void accept (Visitor v) {
        V. visit (this);
        // the condition
        e. accept (this);
        // the two branches
        s1. accept (this);
        s2. accept (this);
    }
}

public class RegenerateVisitor implements Visitor {
    public void visit (AddExpr e) {
        print ("(");
        e. e1. accept (this);
        print (" + ");
        e. e2. accept (this);
        print (" )");
    }

    public void visit (MulExpr e) {
        print ("(");
        e. e1. accept (this);
        print (" * ");
        e. e2. accept (this);
        print (" )");
    }

    public void visit (IfStmt s) {
        print ("if (");
        s. s.cond. accept (this);
        print (")");
        s. s1. accept (this);
        print ("else");
        if (s. s2 == null) print ("{3}"); else s. s2. accept (this);
    }

    public void visit (CompileStmt s) {
        println ("{3}");
        s. s1. accept (this);
        s. s2. accept (this);
    }
}
public void visit (AssignStmt s) {
    s.id.accept (this);  print("=");
    s.expr.accept (this);
    print (";");
}

3. Activation Record
(1) [10p] What are caller-save and callee-save registers?

Callee-Saved registers are saved by callee when entering
the function, restored before returning to main
function. So the register value won't change before and
after calling a function.

Caller-Saved register may change because the callee won't
store it in stack. It will be suitable for variables which
won't use after function call.

(2)[10p] Consider the following code snippet. Assume we put x and y into registers.
Please determine what kind of registers should be used, why?

foo(int option) {
    int x, y;
    x=2;
    if (option==1)
        y=x-2;
    else
        y=x+2;
    A ();
    B ();
    return y;
}

<table>
<thead>
<tr>
<th></th>
<th>x used</th>
<th>y used</th>
</tr>
</thead>
<tbody>
<tr>
<td>function call</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since x is no longer
used after function call,
the value doesn't matter if
A() or B() changed the
registered it or not.

But y should have the
same value at (A).

So:  x - caller-saved.
     y - callee-saved.
4. Translation to intermediate code.

(1) \( t_0 \leftarrow (A[10] > 0) \) (A is a global variable and \( x \) is a local variable, please **DO NOT** use \( \text{unEx}() \), \( \text{unCx}() \), or \( \text{unNn}() \). In other words, you may have to use primitives such as \( \text{CJUMP} \))

\[\begin{align*}
\Gamma_1 & \leftarrow (t_0) \\
\Gamma_2 & \leftarrow (t_0 > 0) \\\\text{(if expression, will use FSEGA t return \( t_1 \))}
\end{align*}\]

\[\begin{align*}
\text{MOVE} & (\text{TEMP}(r_1), \\
\text{MEM} & (\text{BINOP}(+, \text{MEM}(A), \\
& \quad \text{BINOP}(\#, \text{CONST}(10), \text{CONST}(w))) \\
& \text{ CJUMP } (\#, \text{TEMP}(r_1), \text{CONST}(0), \text{true}, \text{false}); \\
& \text{LABEL}(t); \\
& \text{MOVE} (\text{TEMP}(r_1), \text{CONST}(11)); \text{LABEL}(z) \\
& \text{MOVE} (\text{TEMP}(r_1), \text{CONST}(11)); \text{LABEL}(p); \\
& \text{LABEL}(1); \\
& \text{LABEL}(2) \}
\end{align*}\]

(2) \( t_0 \leftarrow (A[10] > 0) \) (s) (A is a global variable, \( s \) represents a statement. You can use \( \text{unEx}() \), \( \text{unCx}() \), or \( \text{unNn}() \) if needed)

\[\begin{align*}
\text{LABEL}(\text{while}); \\
\text{CJUMP} & (\#, \text{MEM}(\text{BINOP}(+, \text{MEM}(A), \\
& \quad \text{BINOP}(\#, \text{CONST}(10), \text{CONST}(w))))) \\
& \text{LABEL}(1); \\
& \text{LABEL}(2); \\
& \text{LABEL}(\text{false}); \\
& \text{S. unNn}(); \\
& \text{JUMP}(\text{while}); \\
& \text{LABEL}(\text{w.end}); \\
& \text{)