1. LR Parsing

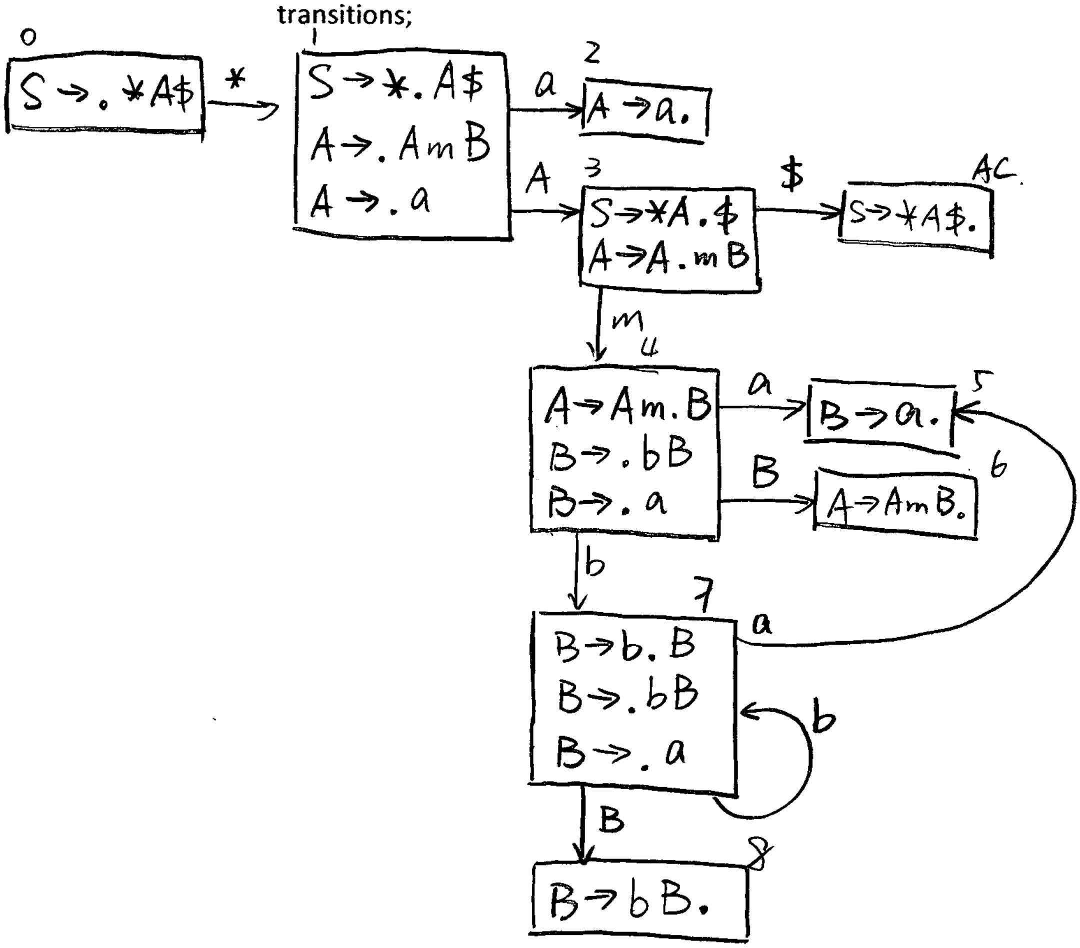
Given the following grammar:

S -> *A

A -> AmB | a

B -> bB | 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)

State	a	b	m	*	$	A	B
0				S1			
1	S2					g3	
2	r3	r3	r3	r3	r3		
3			S4		AC.		
4	S5	S7					g6.
5	r5	r5	r5	r5	r5		
6	r2	r2	r2	r2	r2		
7	S5	S7					g8
8	r4	r4	r4	r4	r4.		

Gramma Rules : (r_k means reduce using rule k) . AC = Accept.

1. $S \Rightarrow *A$.
2. $A \Rightarrow A m B$
3. $A \Rightarrow a$
4. $B \Rightarrow bB$
5. $B \Rightarrow a$.

(3) [10p] Parse the input string ***ama** by filling the following parsing table. The number 1 in the stack column indicates the first state is 0

Stack	Input	Action
0	*ama$	[0, *] = S1.
$0 *^1$	ama$	[1, a] = S2
$0 *^1 a^2$	ma$	[2, m] = reduce: A→a.
$0 *^1 A$	ma$	[1, A] = g3.
$0 *^1 A^3$	ma$	[3, m] = S4
$0 *^1 A^3 m^4$	a$	[4, a] = S5.
$0 *^1 A^3 m^4 a^5$	$	[5, $] = reduce: B→a.
$0 *^1 A^3 m^4 B$	$	[4, B] = g6.
$0 *^1 A^3 m^4 B^6$	$	[6, $] = reduce: A→AmB
$0 *^1 A$	$	[1, A] = g3.
$0 *^1 A^3$	$	[3, $] = ACCEPT

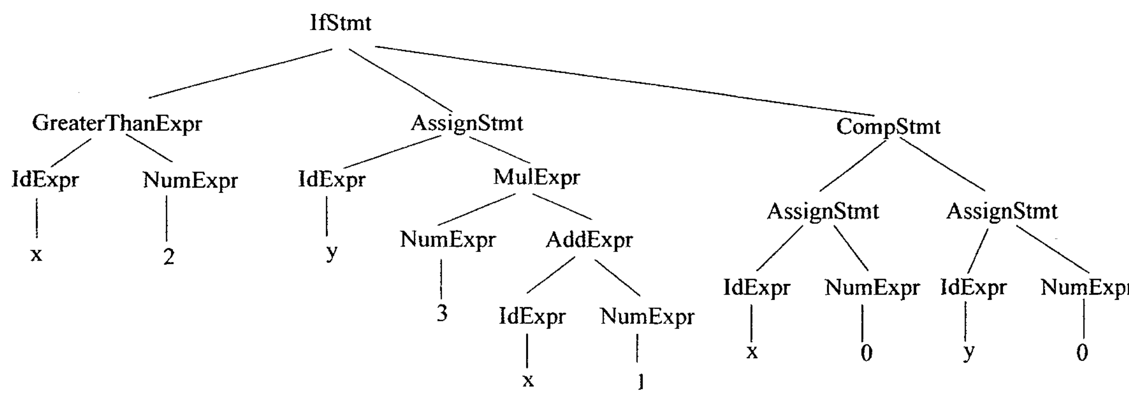
S→*A$

2. Visitor pattern

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

Visitor pattern seperates the data storage and algorithm, providing a more flexible way to process/traversal 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.




```
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 System.out.print & System.out.println respectively.

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

public class IfStmt extends Stmt {
    Expr c;                          // the condition
    Stmt s1, s2;                     // the two branches
    public void accept (Visitor v) {
        v.visit (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.c.accept (this); print (")");
        s.s1.accept (this); print ("else");
        if (s.s2 == null()) print("{}"); else s.s2.accept(this);
        println("");
    }
    public void visit (CompStmt s) {
        println ("{"); s.s1.accept(this); s.s2.accept (this);
        println ("}");
    }

    public void visit (IdExpr e){
        print (e.name);
    }
    public void visit (NumExpr e){
        print (e.value);
    }
    public void visit (GreaterThanExpr e){
        e.e1.accept (this);
        print (">");
        e.e2.accept (this);
    }
```

```
public void visit (AssignStmt s) {
    s.id.accept(this);  print("=");
    s.expr.accept(this);
    println(";");
}
```

3. Activation Record

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

Callee-Saved registers are saved by callee when entering the function, and 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 ();   } function call
    return y;
}
```

x used

y used

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 (☆).

So: x - caller-saved.

y - callee-saved.

4. Translation to intermediate code.

(1) [10p] x= (A[10]>0) (A is a global variable and x is a local variable, please **DO NOT** use unEx(), unCx(), or unNx(). In other words, you may have to use primitives such as CJUMP)

* Assume FP is the frame pointer. Assume this is a STATEMENT! W is word size (const)

(If expression, will use ESEQ to return r1)

$r_1 \leftarrow A[10]$

$r_1 \leftarrow (r1 > 0 ?)$

$x \leftarrow r_1$

```
Nx(
  MOVE(TEMP(r1),
        MEM(BINOP(+, MEM(A)
                   , BINOP(*, CONST(10), CONST(W))
  )));
  CJUMP(>, TEMP(r1), CONST(0), t, f);
  LABEL(t);
  MOVE(TEMP(r1), CONST(1)); JUMP(Z);
  LABEL(f);
  MOVE(TEMP(r1), CONST(0));
  LABEL(Z);
  MOVE(MEM(
   BINOP(+, TEMP(FP), OFFSET(x))
   ), TEMP(r1)
  );
  );
```

(2) [10p] while (A[10]>0) { s } (A is a global variable, s represents a statement. You can use unEx(), unCx(), or unNx() if needed)

```
Nx(
  LABEL(while);
  CJUMP(>, MEM(BINOP(+, MEM(A),
                    BINOP(*, CONST(10), CONST(W)))),   } A[10]
        CONST(0), true, wend);
  LABEL(true);
  s.unNx();
  JUMP(while);
  LABEL(wend);

);
```