1 Analysis & Interpretation

1. Consider the following Jex program:

```
int x;
int f (int x) {
    int z;
    String y;
    /***/
    x = x + 1;
    int y;
}
x = f(5);
```

   (a) Give a graphical description of the symbol table (Scope and Symbol objects) of the symbol table at the point marked /***/

   (b) In the next line the name analysis looks up the identifier x. What Scope and Symbol objects does it look at, in which order?

   (c) In the line after that it has to enter a new symbol for y. What will happen?

2. Consider the following Jex program:

```
int x;
int g (int x, int y) {
    return y;
}
int h (int z, int y) {
    z = x + y;
    /***/
    return z;
}
```

   (a) Give a graphical description of the symbol table (Scope and Symbol objects) of the symbol table at the point marked /***/

3. Give an abstract syntax tree of the following program including the symbols computed in analysis. From the attributes of the Symbol only give the name.

```
boolean x;
boolean g (int y) {
    String z;
    x = x + y;
    return true;
}
```

4. Consider the following program:

```
import java.io.*;
int x; x = 1;
int u; u = 3;
int h(int z) {
        int k;
        System.out.println(x);
        System.out.println(z);
        x = u + z;
        return z;
}
int g(int u, int v) {
        System.out.println(x);
        System.out.println(u);
        x = v + h(u);
        u = 2;
        return v;
}
System.out.println(g(3, h (4));
System.out.println(x);
System.out.println(u);
```

(a) What does the program print?

(b) What are all the environments (Global variables , environments of g and h) when h is called?

5. Consider the Jex fragment. What happens during interpretation? Which of the caseXXX functions of the visitor are called? What are the values of storeVal at the moment of the call and the values of val, and isReturn at the moment of the return?

   (a) `a.x = 7 * a.x;` (Assume a.x being set to 3)

   (b) `if (true) return 3; else x = x + 1;`

   (c) `y = System.out;`

   The solution could look like

```
call caseAssign (storeVal = null)
        call caseOperation (storeVal = null)
              ...
        return caseOperation (val = JexBasic(21), isReturn = false)
        ...
return caseAssign (val = null, isReturn = false)
```

6. The following is an incorrect implementation of the while case in the interpreter. It has two mistakes.

```
public void caseWhile(Tree.While tree) {
    boolean finished = false;
    while (!finished) {
        JexValue b;
        b = interpret(tree.expr);
        if (!b.getBoolean())
            finished = true;
        else
            val = interpret(tree.body);
    }
}
```

(a) What are the mistakes?

(b) Give for each of the mistakes a Jex program, where the mistake would show up.

(c) For each program explain what would happen in the correct and in the incorrect version.

7. Assume we want to introduce a new operator ^^ for exclusive or into our language Jex.

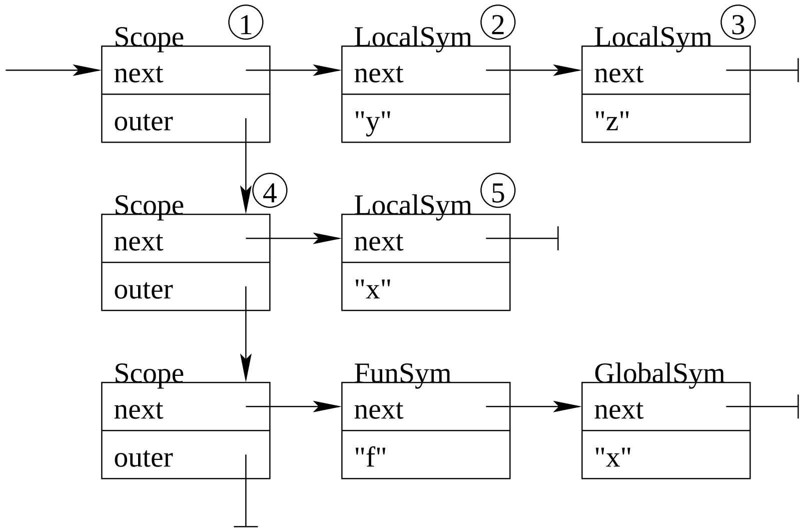
   (a) Give the code fragment that you need to add to caseOperation

   (b) Where else in the compiler would you have to make changes?

   (c) What would those changes be?

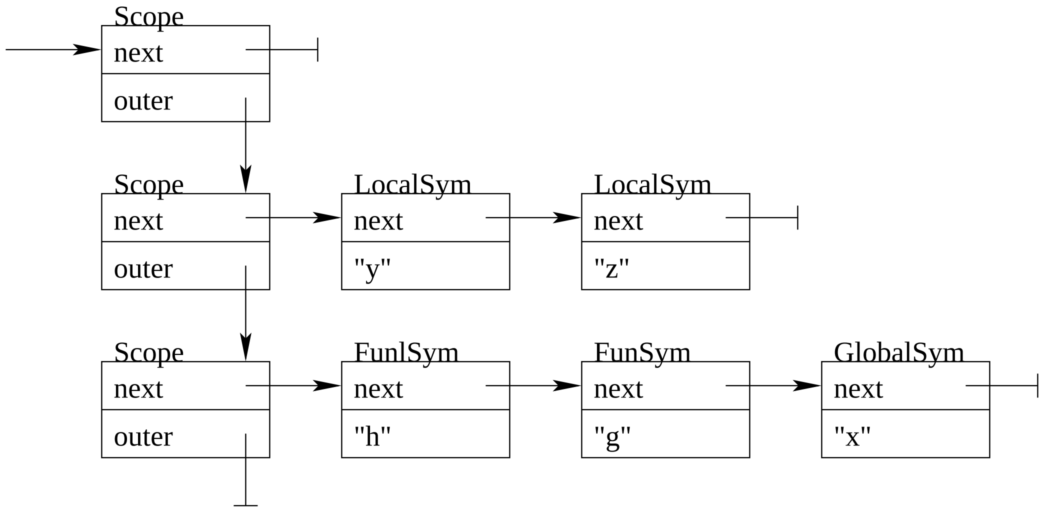
2 Solutions

1. Solution:

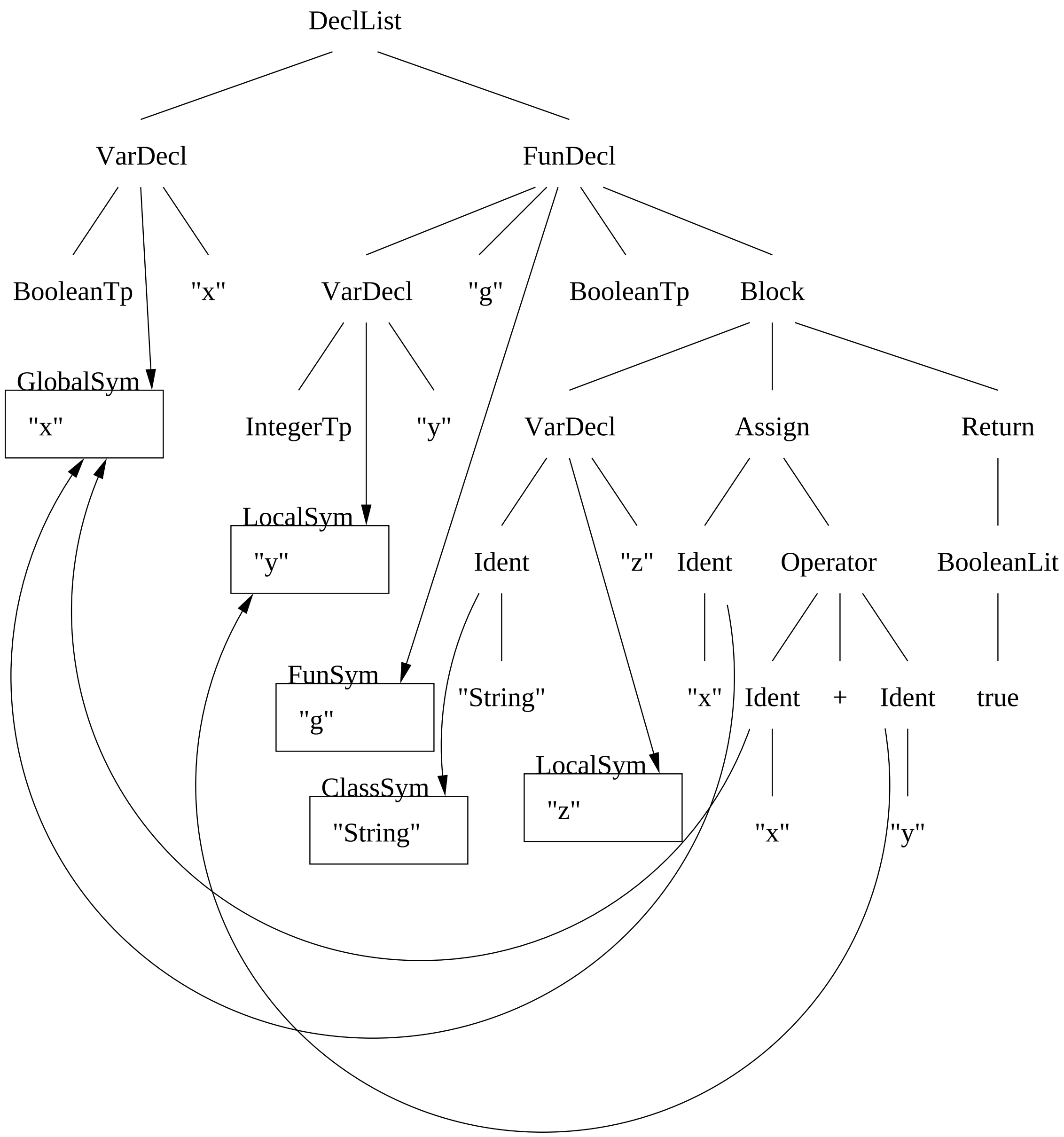


It will find, that there is already a symbol for `y` in the actual scope an will give an error message.

2. Solution:



3. Here is the tree:



4. This is how the variables are set:

global					
x	u				
1	3	call h			
		k	z		
			4		
7					
		call g			
		u	v		
		3	4		
				call h	
				k	z
					3
6					
7		2			

The actual printout is 1 4 7 3 7 3 7 2, but with newlines instead of spaces.

5. (a) a.x = 7 ∗ a.x;

```
call caseAssign (storeVal = null)
    call caseOperation (storeVal = null)
        call caseNumLit (storeVal = null)
        return caseNumLit (val = JexBasic(7), isReturn = false)
        call caseFieldAccess (storeVal = null)
            call caseIdent (storeVal = null)
            return caseIdent (val = JexObject(..), isReturn = false)
        return caseFieldAccess (val = JexBasic(3), isReturn = false)
    return caseOperation (val = JexBasic(21), isReturn = false)
    call caseFieldAccess (storeVal = JexBasic(21))
        call caseIdent (storeVal = null)
        return caseIdent (val = JexObject(..), isReturn = false)
    return caseFieldAccess (val = null, isReturn = false)
return caseAssign (val = null, isReturn = false)
```

(b) **if** (true) **return** 3; **else** x = x + 1;

```
call caseIf (storeVal = null)
    call caseBooleanLit (storeVal = null)
    return caseBooleanLit (val = JexBasic(true), isReturn = false)
    call caseReturn (storeVal = null)
        call caseNumLit (storeVal = null)
        return caseNumLit (val = JexBasic(3), isReturn = false)
    return caseReturn (val = JexBasic(3), isReturn = true)
return caseIf (val = JexBasic(3), isReturn = true)
```

(c) y = System.out;

```
call caseAssign (storeVal = null)
    call caseFieldAccess (storeVal = null)
        call caseIdent (storeVal = null)
        return caseIdent (val = JexClass(java.lang.System), isReturn = false)
    return caseFieldAccess (val = JexObject(System.out), isReturn = false)
    call caseIdent (storeVal = JexObject(System.out))
    return caseIdent (val = null, isReturn = false)
return caseAssign (val = null, isReturn = false)
```

6. (a) 1. A return in the body of the while is not considered.
   2. It is not checked, whether the condition is really a boolean value.

   (b)
   ```
   while (true) return 1;
   ```
   ```
   while (2) x = x + 1;
   ```

   (c) For the first program the incorrect solution would loop indefinitely (it does not check for the return. A correct version should return from the function with result 1.

   For the second program the incorrect solution would raise an error, which would terminate the interpreter. The correct program would raise a JexException, which would be caught by the interpreter-loop and an appropriate error message would be given to the user.

7. (a) Here is the code fragment that you need to add to caseOperation

   ```
   case Tokens.XOR: {
       if (lval.isBoolean() && rval.isBoolean())
           val = new JexBasic(lval.getBoolean() != rval.getBoolean());
       else
           throw new JexException(tree.pos, "type error in ^^");
   } break;
   ```

   (b) In the scanner and parser.

   (c) Scanner(jex.lex): In representation add

   ```
   case XOR: return ("^^");
   ```

   For the patterns add

   ```
   "^^"        { return mkToken(XOR); }
   ```

   Parser(jex.cup): add

   ```
   terminal XOR;
   precedence left XOR;
   ```

   and a rule for Expr

   ```
   |  Expr:l XOR:op Expr:r
       {: RESULT = new Operation(opleft, l, r, Tokens.XOR); :}
   ```