Problem 1

Implemented a simple program analyzer and interpreter for the straight-line programming language. This exercise serves as an introduction to *environments* (symbol tables mapping variable names to information about the variables); to *abstract syntax* (data structures representing the phrase structure of programs); to *recursion over tree data structures*, useful in many parts of a compiler; and to a *functional style* of programming without assignment statements.

It also serves as a "warm-up" exercise in Java programming. Programmers experienced in other languages but new to Java should be able to do this exercise, but will need supplementary material (such as textbooks) on Java.

Programs to be interpreted are already parsed into abstract syntax, as described by the data types in Program 1.5.

However, we do not wish to worry about parsing the language, so we write this program by applying data constructors:

```
Stm prog =
new CompoundStm(new AssignStm("a",
                    new OpExp(new NumExp(5)m
                                OpExp.Plus, new NumExp(5),
                                            OpExp.Plus, new NumExp(3)))m
new CompoundStm(new AssignStm("b",
    new EseqExp(new PrintStm(new PairExpList(new IdExp("a"),
                new LastExpList(new OpExp(new IdExp("a"),
                                OpExp.Minus,new NumExp(1))))),

        new OpExp(new NumExp(10), OpExp.Times,
                    new IdExp("a")))),
new PrintStm(new LastExpList(new IdExp("b"))));
```

Files with the data type declarations for the trees, and this sample program, are available in the directory `$MINIJAVA/chap1`.

Writing interpreters without side effects (that is, assignment statements that update variables and data structures) is a good introduction to *denotational semantics* and *attribute grammars*, which are methods for describing what programming languages do. It's often a useful technique in writing compilers, too; compilers are also in the business of saying what programming languages do.

Therefore, in implementing these programs, never assign a new value to any variable or object field except when it is initialized. For local variables, use the initializing form of declaration (for example, `int i=j+3;`) and for each class, make a constructor function (like the `CompoundStm` constructor in Program 1.5).

1. Write a Java function `int maxargs(Stm s)` that tells the maximum number of arguments of any `print` statement within any subexpression of a given statement. For example, `maxargs(prog)` is 2.

2. Write a Java function `void interp(Stm s)` thats "interprets" a program in this language. To write in a "functional programming" style - in which you never use an assignment statement - initialize each local variable as you declare it.

Your function that examine each `Exp` will have to use `instanceof` to determine which subclass the expression belongs to and then cast to proper subclass. Or you can add methods to the `Exp` and `Stm` classes to avoid the use of `instanceof`.

For part 1, remember that print statements can contain expressions that contain other print statements.

For part 2, make two mutually recursive functions `interpStm` and `interpExp`. Represent a "table," mapping identifiers to the integer values assigned to them, as a list of `id` x `int` pairs.

```
class Table {
    String id; int value; Table tail;
    Table(String i, int v, Table t) {id=i; value=v; tail=t;}
}
```

Then `interpStm` is declared as

```
Table interpStm(Stm s, Table t)
```

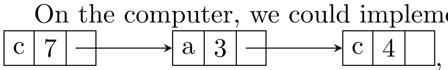
taking a table t_1 as argument and producing the new table t_2 that's just like t_1 except that some identifiers map to different integers as a result of the statement.

For example, the table t_1 thats maps a to 3 and maps c to 4, which we write $\{a \mapsto 3, c \mapsto 4\}$ in mathematical notations, could be represented as the linked list .

Now, let the table t_2 be just like t_1, except that is maps c to 7 instead of 4. Mathematically, we could write,

t_2 = update(t_1, c, 7),

where the update function returns a new table $\{a \mapsto 3, c \mapsto 7\}$.

On the computer, we could implement t_2 by putting a new cell at the headof the linked list:
, as long as we assume that the *first* occurrence of c in the lsit takes precedence over any later occurrence.

Therefore, the `update` function is easy to implement; and the corresponding `lookup` function

```
int lookup(Table t, String key)
```

just searches down the linked list. Of course, in an object-oriented style `int lookup(String key)` should be a method of the `Table` class.

Interpreting expressions is more complicated than interpreting statements, because expressions return integer values *and* have side effects. We wish to simulate the straight-line programming language's assignment statements without doing any side effects in the interpreter itself. (The `print` statements will be accomplished by interpreter side effects, however.) The solution is to declare `interpExp` as

```
class IntAndTable {int i; Table t;
    IntAndTable(int ii, Table tt) {i=ii; t=tt}
}
IntAndTable interpExp(Exp e, Table t) ...
```

The result of interpreting an expression e_1 with table t_1 is an integer value i and a new table t_2. When interpreting an expression with two subexpressions (such as an `OpExp`), the table t_2 resulting from the first subexpression can be used in processing the second subexpression.