Simple translation

The goal of this lab is to build a very simple translator for a mini declarations language. Input looks like:

```
int x;
T t;
```

We will start out emitting text directly with print statements in a listener and then buffer up the strings.

Syntax-directed translation

The simplest approach to building a translator is to print text when we see items of interest from the input. Let's get started with our grammar, which is just:

```
grammar Lang;

file : decl+ ;

decl : typename ID ';' ;

typename : 'int' | ID ;

ID : [a-zA-Z]+ ;
WS : [ \r\t\n]+ -> skip;
```

In a true syntax directed translator, we would add print statements directly in that grammar but of course that makes the grammar specific to a particular application. To avoid this, we will build the usual parse tree and then put print statements in a listener. Technically, this would still be a syntax directed translator, albeit a more flexible one.

Then we need the usual main program that fires up the parser and prints out the tree:

```
String code =
    "int x;\n" +
    "A b;\n";
ANTLRInputStream input = new ANTLRInputStream(code);
LangLexer lexer = new LangLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
LangParser parser = new LangParser(tokens);
ParseTree tree = parser.file(); // start up

System.out.println(tree.toStringTree(parser));
```

On to the translator then. We want to trigger print statements when we see certain input phrases, which we can easily do with a listener. Create a listener called `Gen` an override the appropriate method (you have to figure that out) and insert the following code:

```
String typename = ctx.typename().getText();
String varname = ctx.ID().getText();
if ( isClassName(typename) ) {
    System.out.println(typename+" *"+varname+";");
}
else {
    System.out.println(typename+" "+varname+";");
}
```

Then we need the following method since we don't have type information:

```
/** Pretend we have type information */
public boolean isClassName(String typename) {
    return Character.isUpperCase(typename.charAt(0));
}
```

Have the main program walk the parse tree using this listener. The output you should get is the following.

```
(file (decl (typename int) x ;) (decl (typename A) b ;))
int x;
A *b;
```

Buffering output elements

Syntax-directed translators are very inflexible and tie the order of the output to the order of the input. To decouple that order, we must buffer the output and then emit in the appropriate order later. Order does not matter here, but I want to reinforce this point.

Alter the `Gen` listener so that it adds the output strings to a field:

```
public List<String> decls = new ArrayList<>();
```

Then, alter the main program so that it prints out the declarations after walking the tree:

```
for (String decl : listener.decls) {
    System.out.println(decl);
}
```

You should get the same output but we have a much more flexible translator here. For example, we could reverse the order of the output declarations or even sort them in some interesting way.