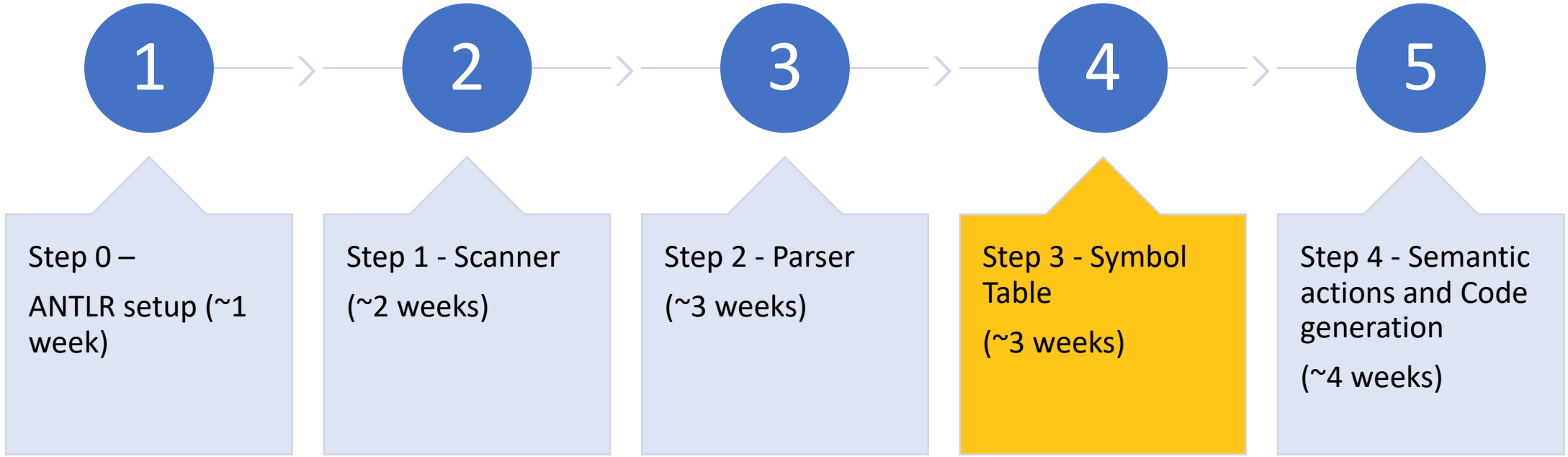
Course Project

Step 3

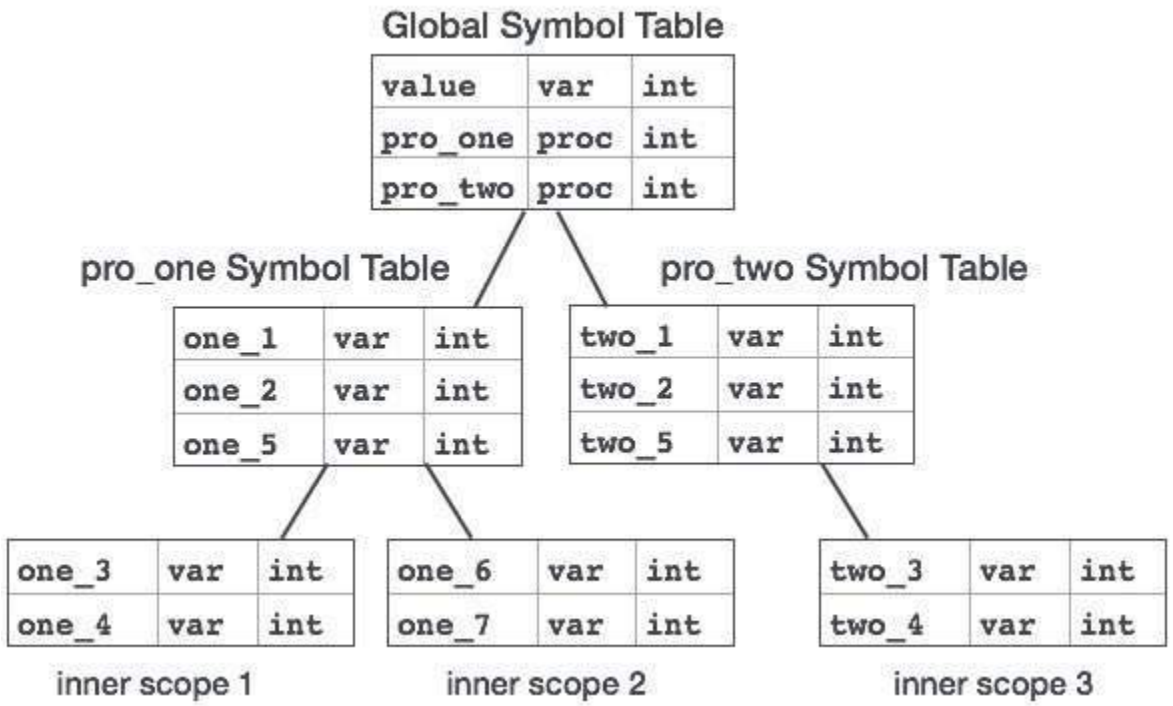
Symbol Table

Project steps



Symbol Table

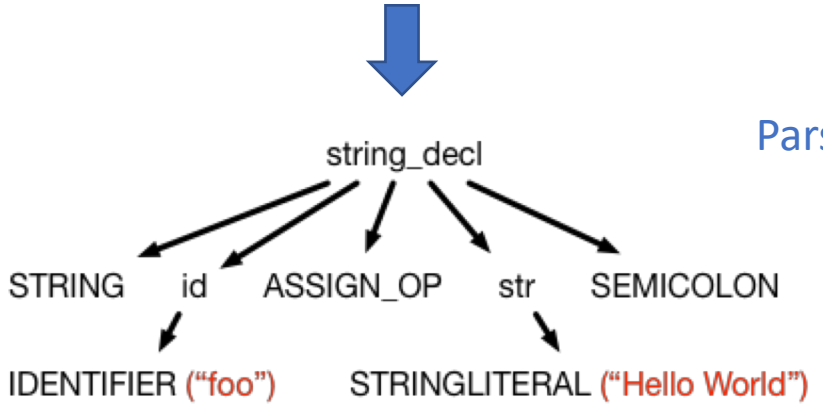
- keeps information about non-keyword symbols (e.g. variables and function names)
- The symbols added to the symbol table will be used later.
- Need to add *semantic actions* to create symbol table entries and add those to the symbol table.



Semantic Actions

- Steps that your compiler takes as the parser recognizes constructs.
- E.g. can create *semantic records* for each of the tokens IDENTIFIER and STRINGLITERAL that
  - record their values ("foo" and "Hello World", respectively),
  - and "pass them up" the tree so that those records are the records for *id* and *str*.

```
STRING foo := "Hello World";
```



Parse Tree

Can then construct a semantic record for string_decl using the semantic records of its children to produce a structure that captures the necessary information for a string declaration entry in your symbol table (and even add the entry to the symbol table).

Symbol table details

- Your task in this step of the project is to construct symbol tables for each *scope* (i.e. there are multiple *scopes*) in your program.
- For each scope, construct a symbol table, then add entries to that symbol table as you see *declarations*.
- The declarations you have to handle are *integer/ float* declarations, which should record the name and type of the variable, and *string* declarations, which should *additionally* record the value of the string.
- Typically function declarations/definitions would result in entries in the symbol table, too, but no need to record them for this step.

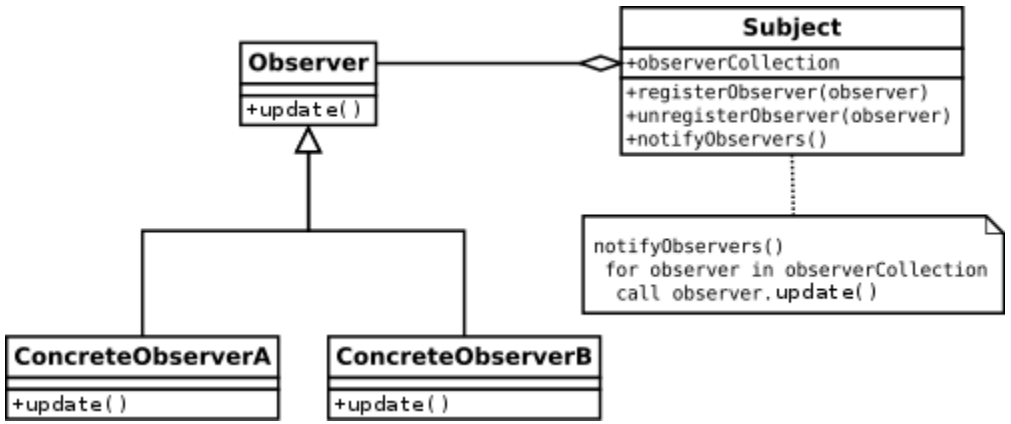
Complete details is in **D2L: Project → Step3 Instructions**

Building the Symbol Table

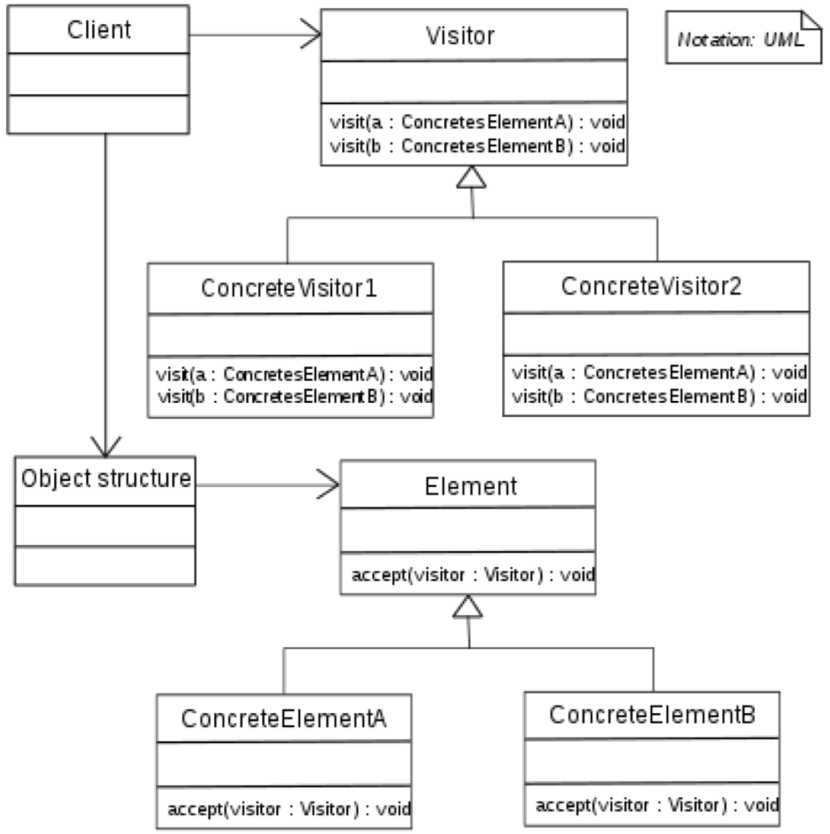
- You should <u>not</u> need to add to your grammar (.g4 file) for this step.
  1. Executing the .g4 file from step 2 will produce a *Listener* and a *Visitor* classes (in-built tree-walking mechanisms)
  2. You can *extend* these auto-generated *Listener/ Visitor* to implement semantic actions.
- Tokens become leaves in the parse tree. The semantic record for a token is always the text associated with that token.
- *Every* symbol that shows up in a grammar rule will be a node in your parse tree

Listener/ Visitor Design Patterns

Observer (Listener)



Visitor



In your report, describe which pattern you used and why.

Data structures

Hash tables / dictionaries

- Fast and easy to use if the language implements them for you

Tree structure

- Relatively fast and easy to implement

List of lists

- Relatively fast and easy to implement

In your report, describe which DS you used and why.

Using ANTLR Listener

```
LittleParser parser =
    new LittleParser(
        new CommonTokenStream(
            new LittleLexer(
                new ANTLRFileStream(args[0]))));

Listener listener = new Listener();
new ParseTreeWalker().walk(listener,
                           parser.program());

SymbolTable s = listener.getSymbolTable();

prettyPrint(s);
```

Using ANTLR Listener (cont.)

```
class Listener extends LittleBaseListener {
    // initialize a symbol table object here
    // may want to have a stack as well...

    @Override
    public void enterFunc_decl(LittleParser.
        Func_declContext ctx){
        // operate on symbol table here
    }

    @Override
    public void exitFunc_decl(LittleParser.
        Func_declContext ctx){
        // operate on symbol table here
    }

    // additional rules and/or helper methods here...

}
```

Using ANTLR Visitor

- Driver class will have a similar structure to the *Listener* example
- Visitor instances must have a *return* value in ANTLR
- This is different from the *Listener*, which performs actions without returning.

What you need to do

- You should define the necessary semantic actions and data structures to let you build the symbol table(s)
- At the end of the parsing phase, you should print out the symbols you found. For each symbol table in your program:

  Symbol table <scope_name>

  name <var_name> type <type_name>

  name <var_name> type <type_name> value <string_value>;

  ...

What you need to do

- Scopes:
  - The global scope should be named "GLOBAL",
  - function scopes should be given the same name as the function name, and
  - block scopes should be called "BLOCK X" where X is a counter that increments every time you see a new block scope.
  - *Function parameters should be included as part of the function scope.*
- *The order of declarations matters!*

Do not need to add anything new to the grammar (.g4) file from Step 2