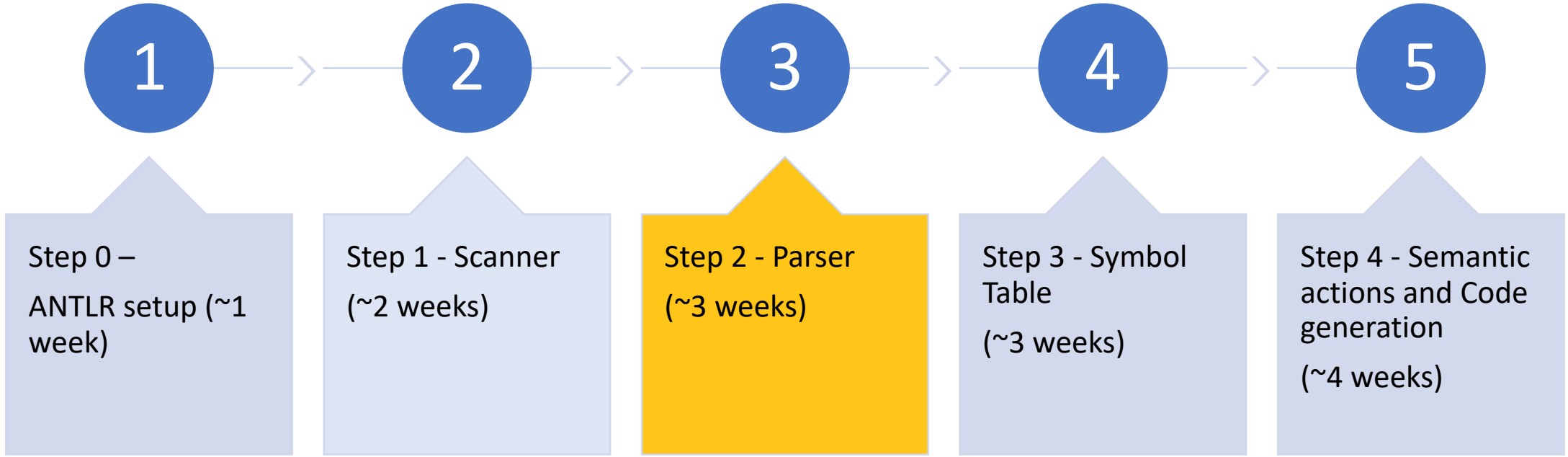
Course Project

Step 2

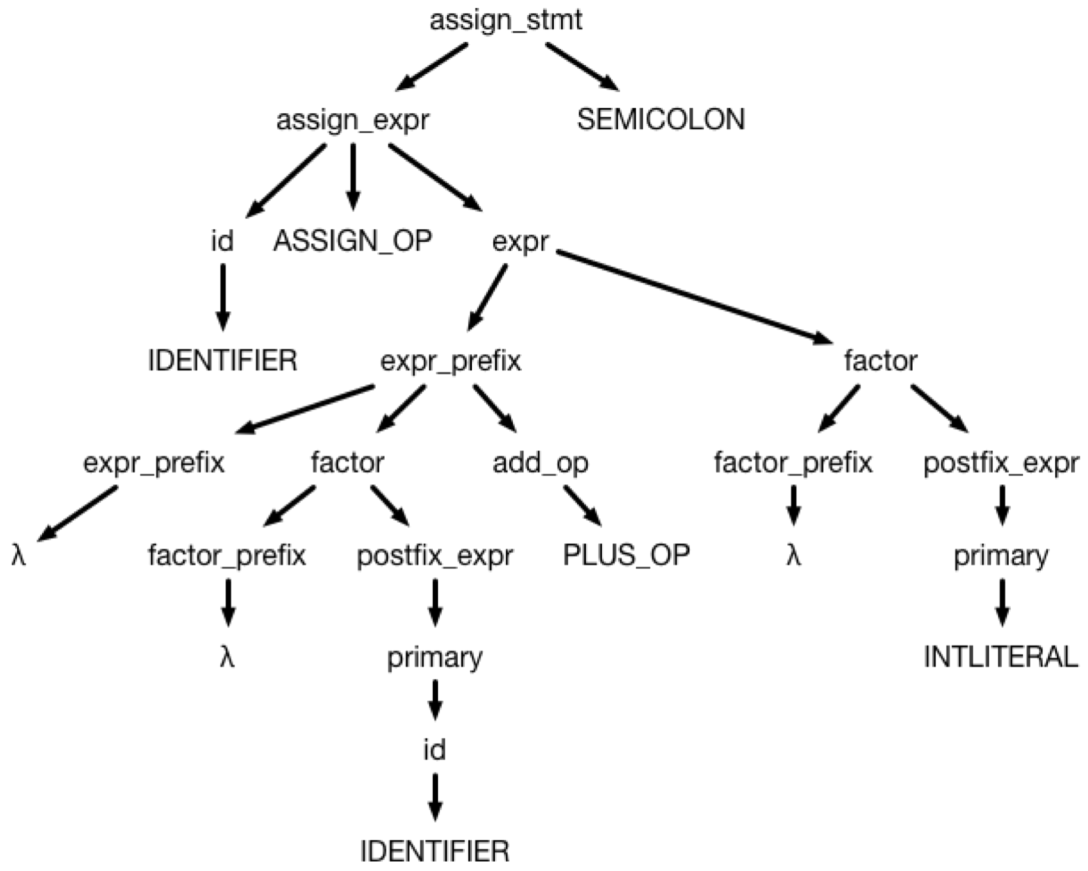
Parser

Project steps



Parser

- The job of a parser is to convert a stream of tokens (as identified by the scanner) into a *parse tree*: a representation of the structure of the program.
- E.g: it will convert: A := B + 4;
  Into a tree that looks something like:

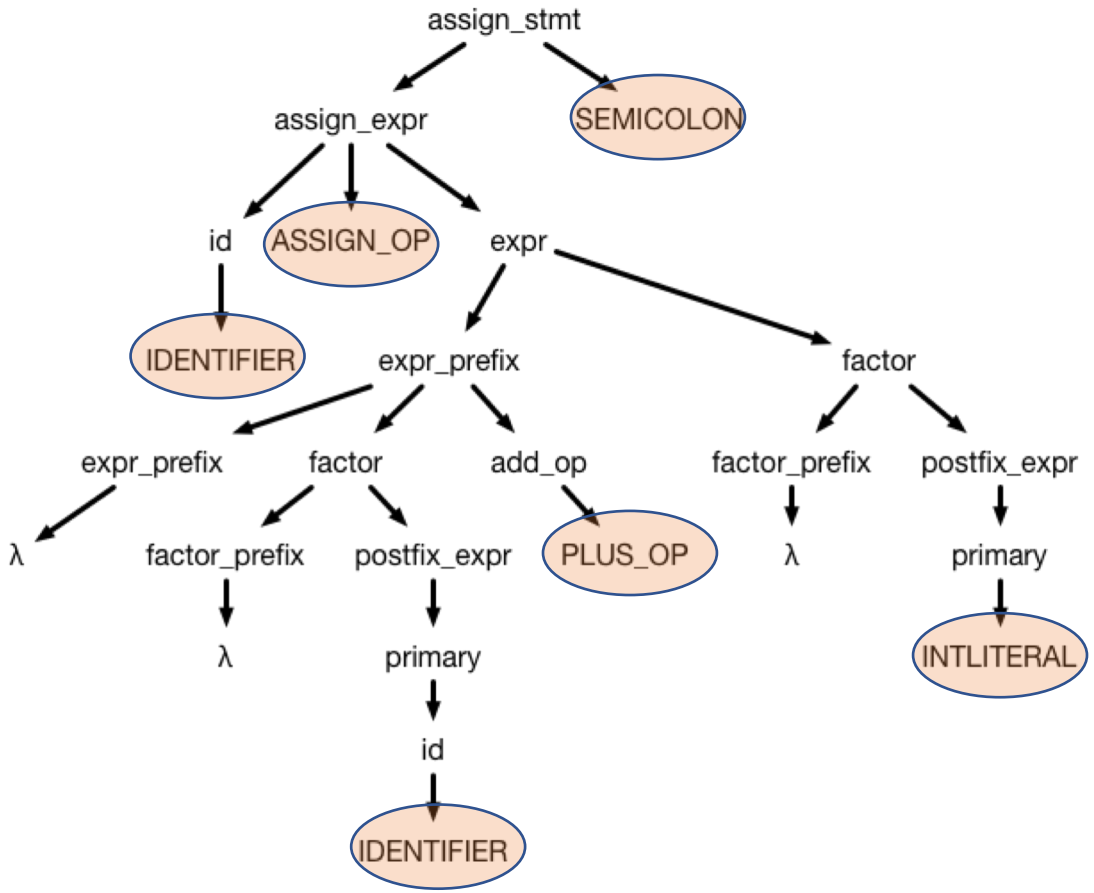


Parser

- The *leaves* of the tree are the tokens of the program.
- If you read the leaves of the tree left to right (ignoring lambdas, since they just represent the empty strings), you get:

  IDENTIFIER ASSIGN_OP IDENTIFIER PLUS_OP INTLITERAL SEMICOLON

  Which is exactly the tokenization of the input program!



Context-free grammar (LITTLE)

```
/* Program */
program             -> PROGRAM id BEGIN pgm_body END
id                  -> IDENTIFIER
pgm_body            -> decl func_declarations
decl                -> string_decl decl | var_decl decl | empty
…
/* Complex Statements and Condition */
if_stmt             -> IF ( cond ) decl stmt_list else_part ENDIF
else_part           -> ELSE decl stmt_list | empty
cond                -> expr compop expr
compop              -> < | > | = | != | <= | >=
…
```

Complete grammar is in **D2L: Project → Step2 Instructions**

CAPS : CAPS is a token (terminal) made up of one or more characters. small case symbols are non-terminals.

Building a Parser

- There are many tools that make it relatively easy to build a parser for a context free grammar (in class, we talk about how these tools work)
- All you need to do is provide the context-free grammar and some actions to take when various constructs are recognized.
- The tool we are using is ANTLR. You should define your grammar in the same .g4 file in which you defined your lexer.
  1. Executing that `.g4` file will produce both a Lexer class and a Parser class.
  2. In your `main` file, rather than initializing a lexer and then grabbing tokens from it (as you may have done in step 1), you instead initialize a lexer, initialize a `CommonTokenStream` from that lexer, then initialize a parser *with the CommonTokenStream you just created*.
  3. You can then call a function with the same name as your top-level construct (probably `program`) on that parser to parse your input.

What you need to do

- Using the grammar for LITTLE given, you need to have your parser parse the given input file and print
  - Accepted if the input file correctly matches the grammar, and
  - Not Accepted if it doesn't
- Testcases are provided.
- A few *hidden* testcases may be used.
  - But, they are very similar to open testcases (they test the same functionalities)
  - If you can get 100% for open testcases → most likely you will get 100% for hidden testcases.