The missing files ourPython.tab.c and ourPython.tab.h and an object file (not sure what the object file is) are created by the previous line "bison -d --debug ourPython.y", and there's something called "MakeFile" in the zipped starter code as well (not sure how to use it but it might be a command perhaps?)

recommended flow/plan: it is recommended that only the ourPython.y file and the ourPython.lex file should be modified, and leave all the other files the same. Look for the commented lines that say "//erase this line and perhaps put [something] here". In the .y file, define the % union, the types, the tokens, then go to the .lex file and give regular expressions for your tokens. Then go back to the .y file and give the parser rules and their actions. you don't have to implement all of the grammar of python , just just implement the ones in the instructions which is most likely #1 and #2. you don't have to implement function calls or loops or anything not listed in the instructions. And for #2 there's 'string' in single quotation marks and "string" in double quotation marks, so both of those count for different things so they might need different definitions (or something like that)

the 1.py 2.py 3.py 4.py 5.py are tests after you're done (i think) see the bottom of the instructions

bison -d --debug ourPython.y <---this line makes the later files (ourPython.tab.c ourPython.tab.h, then the object)

```
compound_stmt: if_stmt
                 {
                  $$ = $1;
                }
                 | while_stmt
                {
                 $$ = $1;
                };
```

```
%union
{
	Expression* exprPtr_;
	statement* statePtr_;
}
%token IF ELIF ELSE
%token colon
%type<statePtr_> if_stmt
%type<exprPtr_> test
```

test is $2 and suite is $4 <-

proto_if_stmt: IF test COLON suite

```
                {
                    $$ = new IfthenElseStatement($2,$4)
                }
                | proto_if_stmt ELIF test COLON suite
               }
```

Example:

Before translating to bison (https://docs.python.org/3/reference/grammar.html)

```
if_stmt:    'if' test ':' suite
            ('elif' test ':' suite)*
            ['else' ':' suite]
```

After translating to bison (part of the answer, not the whole thing I believe):

```
proto_if_stmt: IF test COLON suite
                {
                    $$ = new IfthenElseStatement($2,$4)
                }
               | proto_if_stmt ELIF test COLON suite
               }
                    $$ = $1;
                    $$-> appendElif($3,$5);
```

What should $$ be equal to?
(Look in Statement.h?)

$1 seems to be equal to proto_if_stmt because he is the root of the if statement?

the whole purpose of this is to find a value for $$
so $$ = $1;

first determine which ones are terminals
then determine which ones are non-terminals
then determine which ones are tokens?

after translating to bison:

```
%union
{
      statement* statePtr_;
      %token
      %token<statePtr_> if_stmt>
}
```

suite and test need to be read but there's no data in them (so they're not terminal/non-terminal?)
only change .lex and .y file

we were looking at the Statement.h file where it says "appendElif appendElse" section while we were working on this: (translated to bison)

```
if_stmt: proto_if_stmt
       {

       }
     |   proto_if_stmt ELSE COLON suite
       {
         $$=$1;
         $$->appendElse($4)
       }
```

True and False are capitalized
have atom also handle parenthesis
for atm_expr: [AWAIT] atom trailer*
ignore trailer*
just say
atm_expr: [AWAIT] atom

before translating to bison rules:

factor: ('+' | '-') factor | power

(the token for this after the %union would be:
%token PLUS)

after translating to bison rules:

```
factor: PLUS factor
        {
              $$=new UnaryExpression (PLUS_OP, $2);
       }
     | power
       {
         $$=$1;
       }
       There would be another set of curly braces for MINUS so the above is not the full answer
      {

       }
```

(we looked at #2 to determine what it is and determined that it was an Expression, so we opened Expression.h to look at)

(then we chose to define "power" instead of choosing factor (or something like that))

they meant it like factor:(('+'|'-') factor) | power <- what they meant by the before bison translation