We are now at type checker stage
We looked at lexical analysis, syntax analysis
We finished the front end, which processes the input language

binaries.tar
the “binaries” was not working exactly as the instruction was.
go to the webcourses and redownload new binaries.
people who didn’t complete their previous projects can still get points in future projects.

Symbol table and the annotated AST

- AST is the core data structure of the program that you can do lots of program analysis.
- In annotated AST, we have an extra field “scope”. Because of static scoping we can add variables with the same name but with different scopes or types.
- Extra bookkeeping will make code generation step easier. You can walk through the AST and annotate it but using symbol table makes it very simpler. You can easily search or add symbols.
- Symbol table: all the symbols in your program and their types.
- Symbol table is like a tree itself - nested according to scope info.
- Symbol table only records identifiers in the language.
- AST keeps the entire structure of the language.
- Symbol table is used to check if the types are used correctly.

Question: If we do not have any variables at all, that program will be completely safe? Yes
Just “write x” is not a legal program in PL0, because we have to define x first.
It will have no parser error but type error.

In the static typing rules, the symbols should be used with the types they are defined as.
Type inference: we have languages that can infer the type. When we open java compiler and python compiler and c compiler, they are different in the way that they have implemented syntax analyzer and typechecker.
For example, in python it is not required to specify the type of the variables. but in java and c it is necessary to declare the type when we define the variable.
That is more complicated. We are doing a C-like language that is statically typed

Context-free grammars cannot encode type relationships
Context-sensitive grammars can - which is more complicated than CFGs

Question: What part of the language specification tells us about the precedence of the operators?
Grammar defines our order of operations. It is done in parsing stage.

We can hack the grammar so that order of operations are embedded into the grammar
Any string has only one derivation, such that, that defines a tree that has only one way for evaluation.

If it is higher up in the tree, it is going to be evaluated the last
For example, we have expr ::= simpleexpr [relop simpleexpr], which will make the relational operations happen last.
Then, other operations are mapped in lower levels.
Lower in the tree means that operation is going to happen first.
If the operation is higher in the tree, it happens last.
We have this grammar:
E->E*E | E+E | id
for string id + id * id there are two ways to derive this string from the grammar. So this is ambiguous grammar.

If you have a PLUS or MULT, these are going to happen before relation operation.

Lots of these things are language design choice. It is good to pick a design and stick with it.

note:
exprlist has been updated from “required” to optional in the GRAMMAR:

Before	After
exprlist := expr{COMMA expr}	exprlist := [expr{COMMA expr}]

Question: Can we infer the grammar from inputs? Possibly yes. Fuzz testing might be a solution for this. You generate mass of amounts of inputs to the program and try it.
One way to improve fuzz testing is using the fact that not all inputs are valid inputs.

Doxygen documentation could be helpful - included in your project repository
It documents all the structs. Use it if the given text files are not sufficient.
For unions, C allocate the biggest data type

We will see how the code layout works in machine level.
Our compiler will figure out ahead of time how much space is needed for variables and functions.
There is nothing in the machine that says “it is a struct”. It is just linguistics we implement at compiler level.

Struct just names a chunk of memory. Instead of giving some memory offset to reach some element, you give the name of the element.
Struct allows you to have multiple fields. Union allows you to have optional values.

You can generate code from yml files.

Debugging:

- First thing for debugging is to see what's broken: run the program
- If you don't have a sense of where the problem is, try to come up with a very simple test case that reproduces the problem, so, minimize the debugging effort
  There is research on test case minimization.
- Debugging requires the domain knowledge of the code you are writing and the test case
- Use an editor that allows you to do searching: if you are scrolling in your code, it wastes your time. Search tools makes you more efficient.
- Don't keep things in your mind: find the reference and use the reference. Don't try to memorize stuff.

Result of debugging: (Example)
It might be the problem with creating AST.
write_expression = FuncionFactor
under FunctionFactor we have function_symbol which we don't use, and also function_name.
So the problem might be with function_name

First, let's check the grammar for factor:

factor := IDENT [LPAREN explist RPAREN]

1. factor := IDENT
   or
2. factor := IDENT LPAREN exprlist RPAREN

Factor could be also written as:
IDENT | IDENT LPAREN exprlist RPAREN
according to ast.md (AST Nodes for Parsing) we have:

1. `VariableFactor(variable) for IDENT`
2. `FunctionFactor(function_name, function_parameters) for 'IDENT LPAREN exprlist RPAREN'`

Notes:

- The program is never wrong. It does exactly what it is instructed to do. The problem is that your specification what the program should do is wrong.
- Remember the postcondition and precondition, stick with them.
- Any parsing function should leave the last token it processed.
  In parsing process, we need to point to the last token of that part in the function.

- Write unit tests, make sure that every little part of the program works.
- Whenever you write a if statement, always think of its negation and make sure that you cover the negation of that condition.
- First try to make the parse tree working and then start creating AST by using that.
- In order to create AST(adding nodes, struct,..), we need to create parsing part first (next(), previous(), ensure_token(),..).