Debugging is very important. Next time we will have open Q&A on debugging.

Review:
Lexer: inputs stream of characters and outputs stream of tokens.
Syntax analyzer: inputs token and outputs abstract syntax tree (parse tree).
Semantic analyzer: takes an AST and outputs yes or no.

- Typechecker annotates type related info on AST and checks if this program is correctly typed in this language or not. Also outputs symbol table.
  Why symbol table? Having a symbol table makes creating p-code very easier.

Notes:
Read markdown files: they cover most of the questions of yours
Use the API: it is much easier that way

- Read typechecker.md, which includes detailed information on what is below:
  Datatype.h: Given DataType struct, helps us to check the type of the data (isBool(), isInt(), equalTypes() etc..)

  Symtab.h: Symbol table related functions. Adding scope & getting parent scope, adding variable or function to symbol table, searching symbol from symbol table.

- Read type_specification.md:
  The type related rules of the language and restrictions. What is a correctly typed program?
  Human language description of the type specification of the language

  Example rules:
  - if you don't declare the type you cannot do the operations correctly in pl0.
  - formals are essentially local variables. so we will have addVariable.
  - int is the correct type for PLUS, MINUS, MULT, DIV, MOD operations
  - bool is the correct type for comparing operations such as LT, LTE, ..
    also bool is the type for AND, OR
  - For binary expressions, both left and right operands should be of the same type.
  - AND and OR are (bool, bool) -> bool, which means they take two operands of type bool and returns bool.

  Scoping:
  Because of the static scoping, if you cannot find a symbol at the current scope, you need to trace the parent scopes. **searchSymbol** function handles it for you.
  Ershov numbering:
  Ershov numbering is pretty simple algorithm to figuring out which register to use for which expression in an optimal way.

Statement:
type of assignment should match the expression type.
for call we should declare the type of the function and also its variables.

We are going to do an example on **syllabus/project/tests/function.pl0**

```
var x : int
var result : int
function f(x : int) : int
  begin
    return x
  end
begin
  x := 2
  result := f(x)
  write result
end
```

Three main pieces of typechecking process: **AST, symbol table, current scope**

- For typechecking, the first thing we need is the **AST**. We will fill the extra fields of the AST while typechecking process. We will record information about the symbols and the scope.
- We also need **symbol table** and **current scope**, which we will modify and use during typechecking.
- We need **symbol table** to record declarations. We can save the scope and restore scope as well.

We have the typed AST and symbol table in the tests directory:
syllabus/project/tests/function.types

1. translationunit → **addScope()** → **GLOBAL (current scope)**
2. vardecl is a list and you iterate over this list
   vardec → **addVariable()** → symbol table →

name	type	scope
x	int	GLOBAL

3. addidentifier (**addVariable()**) for result

name	type	scope
x	int	GLOBAL
result	int	GLOBAL

4. we have a function f, so we have **addFunction()**:

name	type	scope
x	int	GLOBAL
result	int	GLOBAL
f	(int)->int	GLOBAL

add symbol field to the AST node as well. **symbol : GLOBAL.f**
For functions, in addition to addFunction(), we also need to use **addScope**() function since functions create a new scope. So **current scope: GLOBAL.f**

5. we have another variable x in the function f. so we call addvariable. and we also add **symbol : GLOBAL.f.x**

name	type	scope
x	int	GLOBAL
result	int	GLOBAL
f	(int)->int	GLOBAL
x	int	GLOBAL.f

Notes:

- Current scope is like a stack of scope. For example, if we add the scope of function f while we are at GLOBAL scope, the current scope will be GLOBAL.f
  So, when we have some variable, which is x in our example, inside that function f, the variable's scope will be GLOBAL.f
  The access path for the variable x is GLOBAL.f.x
- When you are done processing a inner scope, you are going to restore to the previous scope. So we go back to global scope, then block. Now we need to check the global statement.

- Whenever you see a variable factor (varfactor), you need to check the symbol table. That is when you call **searchSymbol**() function.
  searchSymbol() takes the current scope and the name of the variable. Starting from the current scope and possibly going to the parent scopes, it searches symbol at the symbol table.

- At return statement, you check if the return type of the function matches with the expression being returned.
  For return statement, we need to look up the entry for the function since you need to know the return type of the function.
  Then, you will check if the type of the expression being returned matches with the return type.

- For assignment statement, make sure if the type of the variable you are assigning the value to matches with the type of the value. For this, you need to have a symbol table lookup using **searchSymbol**() to get the type of the variable you are assigning a value to.
  - GLOBAL.x
  - datatype = int
  - Then, we compare against the type of the right-hand side expression. Here, you need to use **equalTypes**() function.
  - numberfactor type is always int (hardcoded to be INT).

- To know the type of the functionfactor, you need to know the return value of the related function. Hence, you need a symbol table lookup.

- The legal types for writestatement are INT and BOOL. if it is not we will have typeerror.

Another example:
type checking binary operation
numberfactor type is always int
datatype : int
numberfactor
datatype : int
the data type for binary operation is also int
binary : int

The example walk over the AST is available on the syllabus repository.
The good part is you can program it and you do not need to do it by yourself.
You can see the symbol table which is printed on top of the AST.
It is a little different from what we showed here.

How to do ershov numbering allocation?
how many registers do you have? 16, 32, it depends..
Ershov numbering is an algorithm to use the registers in an optimal way.

Question
If you have constant value you only need one register to store it.
We have two values and two registers. how many registers do we need to do the operation?
We could use 3 registers, but we know that we are not going to use these values later, so we can reuse one of the registers, so ershov number is 2.

child ershov numbers are the same and both are 1 → parent ershov number = child ershov number + 1 = 1 + 1 = 2
one child ershov number is 2 and the other one's is 1 → parent ershov number = max(child ershov numbers) = max(1, 2) = 2 So 2 registers are required to do the operation based on ershov algorithm.

This example is a good demonstration of ershov numbering algorithm.

```
begin
# example of register allocation via Ershov numbering
# each number is a leaf with label 1
# each operation is an inner node with a label according to the Ershov numbering rules
write
  ( 3    # 1
      +    # left == right, left + 1 = 2
    1    # 1
  )
  /    # left == right, left + 1 = 3
  ( 2    # 1
      *    # left != right max(left, right) = 2
    (  5    # 1
         -    # left == right, left + 1 = 2
       4    # 1
    )
  )
end
# (3 + 1) / (2 * (5 - 4))
# (3 + 1) / (2 * 1      )
# 4       / 2
# 2
```

It takes a particular expression and figures out the minimum number of registers required to compute that expression.

Ershov numbering algorithm:
Ershov algorithm has a fairly simple rules set:
1- Leaves on the tree only requires 1 register, hence, leaves get ershov number 1.
How do we define it for the intermediate nodes?
You need to store the operands and the result. However, since we don't need the operands after the operations anymore, hence, we can use their register for the intermediate value.
2- If two children nodes have the same ershov number, parent tree ershov number is that number + 1:
3- If two children nodes have different ershov numbers, the ershov number of the parent tree is MAX(ershov_left, ershov_right)

Coding part of Typechecker:

- For typechecker, you have one function for each type of the nodes of the AST, visit functions.
- Make sure that you use checkExpressionList (no need to write it yourself)
- Classroom exercise: coding typechecking on unary expressions (code is posted on repository)
- type_error(): If you see an error case, call type_error() function with the appropriate error parameter. The parameters are defined in the c file (#define SOME_ERROR "error string").

how to do type checking on unary operation?

```
static void visitUnaryExpression(struct Expression *node) {
  // given
  visitExpression(node->unary_expression);
  struct DataType *operand_type = node->unary_expression->datatype;
  switch (node->unary_op->kind) {
  case MINUS:
    // int -> int
    if (isInt(operand_type)) {
      node->datatype = getInt();
    } else {
      type_error(EXPRESSION_EXPECTED_INT);
    }
    break;
  case NOT:
    // bool -> bool
    if (isBool(operand_type)) {
      node->datatype = getBool();
    } else {
      type_error(EXPRESSION_EXPECTED_BOOL);
    }
```

```
    break;
  default:
    assert(false);  // not supposed to happen if parser is correct
    break;
  }
}

}
```