Q4: Define the semantic analysis phase used in your miniC compiler?

Ans:

The semantic analysis phase performs these checks:

1. **Type Checking:**
   * **Process:** The analyzer enforces that operations use compatible types. It verifies the type of expressions and assignments against predefined type rules.
   * **Example:** Attempting to add a boolean and an integer is reported as a type error. The system makes sure that operations are performed on compatible types.
   * **Implementation:** The code that performs the check for addition nodes is done as follows, when checking an AST node for addition, the types of the two operands are matched to verify they are both integers:

let check\_add\_type e1 e2 =

match (type\_of e1, type\_of e2) with

|(IntType, IntType) -> IntType

|\_ -> failwith "Type error: Addition requires integers"

1. **Scope Checking:**
   * **Process:** Variables are validated to exist within their defined scope, using a scope tracking mechanism during AST traversal. Out-of-scope variable accesses trigger errors.
   * **Example:** Using a variable that is not declared in the current scope causes a scope error.
   * **Implementation:** A symbol table is implemented, and is used to verify that all variables exist within the current scope as defined by the code. This happens when an AST node that references a variable, is being processed.

let check\_scope env id =

match lookup\_symbol env id with

| Some \_ -> ()

| None -> failwith ("Scope error: Variable " ^ id ^ " not in scope")

1. **Declaration Checking:**
   * **Process:** The analyzer verifies that all used variables, functions and labels are declared before their usage, preventing invalid references.
   * **Example:** If a variable, or function is used without a definition, then it is flagged as a declaration error.  
     \* **Implementation** As an AST is traversed, a symbol table is maintained, which lists all declared variables and functions. If a reference is found that is not present in the symbol table, then a declaration error is flagged during processing.

let check\_declaration env id =

match lookup\_symbol env id with

| Some \_ -> ()

| None -> failwith ("Declaration Error: Undeclared Symbol " ^ id)

1. **Function Call Checking:**

* **Process:** This check validates function calls, ensuring the function exists, has the correct number of arguments, and that the types of provided arguments match the expected parameter types.
* **Example:** If a function is called with the wrong number of arguments or with arguments of an incorrect type, the semantic analyzer will report an error.
* **Implementation:** The semantic analysis module includes a check that uses a lookup function to verify if the function is defined. Then, the number and types of provided arguments are matched to the function's defined parameters.

let check\_func\_call env func args =

match lookup\_function env func with

| None -> failwith ("Function call Error: Function not defined " ^ func)

| Some f ->

if List.length args != List.length f.params then

failwith ("Function call Error: Arguments of function call don't match " ^ func)

(\* Add additional code to check if argument types match function parameters \*)

1. **Return Type Matching:**

* **Process:** This check verifies that the type of the expression returned in a function matches the function's declared return type, as specified in the function definition.
* **Example:** If a function is declared to return an int, but the return statement tries to return a boolean value, a type mismatch error is generated.
* **Implementation:** When a return instruction is encountered, a check is performed between the return type and the type of the expression being returned, as shown in this example:

let check\_return\_type fun\_type exp\_type =

match (fun\_type, exp\_type) with

| (IntType, IntType) -> ()

| (VoidType, VoidType) -> ()

| \_ -> failwith ("Type Error: type of expression does not match function's return type")

1. **Initialization Checking (Optional, but often useful):**

* **Process:** This check ensures that variables are initialized before they are used for the first time, so that the program logic will not be undefined if a variable is used that does not have a default value.
* **Example:** If a variable is read before any assignment has been made, then this would be flagged as a semantic error.
* **Implementation:** The code can keep track of if variables have been written to before being used, by using a simple flag. If a read operation is encountered before a write operation then a semantic error is flagged.

let check\_initialization env id =

match lookup\_symbol env id with

|Some var -> if var.initialized = false then

failwith ("Semantic Error: Variable " ^ id ^ " used before initialization")

else

()

|None -> failwith ("Semantic Error: Undeclared symbol")