3.1 THE NEED OF SEMANTIC ANALYSIS

Semantic Analysis is the third phase of a compiler, performed after syntax analysis. It ensures that a program is semantically correct, meaning it follows the logical rules of the programming language. While syntax analysis checks the structure of the program, semantic analysis checks its meaning, such as correct use of variables, types, and expressions.

It works on the parse tree (or AST) and symbol table to:

Verify type correctness

Ensure proper declaration and scope of identifiers

Validate expressions and function calls

During this phase, the syntax tree is enriched with type and scope information, forming an Annotated Syntax Tree, which is used in later stages like intermediate code generation. **Need**

Even if a program is syntactically correct, it may still be incorrect logically.

Example:

int a;

a = "hello"; // syntactically correct but semantically

wrong Syntax is valid, but type rules are violated.

Thus, semantic analysis ensures:

Correct interpretation of meaning

Consistency in program behavior

Error detection before code generation

3.1.1 Role of Semantic Analyzer

The Semantic Analyzer performs the following major tasks:

Type Checking

Scope Resolution

Symbol Table Management

Function Call Validation

Intermediate Representation Support

Syntax Tree Annotation

Functions

1. **Type Checking**

Ensures that operands of operators are of compatible types.

Example:

int x = 10;

float y = x +

2.5; // valid

Invalid case:

int x = "abc"; // type mismatch

2. **Scope Resolution**

Determines the region in which a variable is accessible.

3. **Type Conversion (Coercion)**

Automatically converts data types when required.

Example:

float x = 10.5;

float y = x * 2; // 2 → 2.0 (implicit conversion)

4. **Function Checking** Ensures:

   Correct number of arguments

   Correct argument types

   Correct return type

Example:

int add(int a, int b);

add(5); // error:

missing argument

5. **Label Checking**

Ensures that all labels used in the program are properly declared and referenced.

6. **Flow Control Check**

Ensures proper use of control flow statements.

Examples:

   break only inside loops or

switch return matches

function type

7. **Symbol Table Handling**

Semantic analysis uses the symbol table to:

Store variable names, types, scope

Track function definitions

Ensure correct usage of identifiers

3.1.2 Semantic Errors

Semantic errors are detected after syntax analysis. Common errors include:

1. Typ

e

Mismatch

int x =

"hello";

2. Und

eclared

Variables x

= 10; // not

declared **3.**

Multiple

Declaratio

ns

int x;

int x; // redeclaration

4. **Reserved Keyword Misuse** int while = 5;
5. **Incorrect Function Usage**

int add(int a, int b);

add(1); // wrong number of arguments

1.3.3 Static vs Dynamic

Semantics Static Semantics

Checked at compile time

Includes type checking, scope rules, declarations

Helps detect errors early

Dynamic Semantics

Checked at runtime

Defines actual behavior of program execution

Examples:

Division by zero

Null pointer access

3.1.4 Annotated Syntax Tree

During semantic analysis, additional information is added to the syntax tree.

This helps in intermediate

code generation **Example of**

Semantic Analysis float x =

10.1;

float y = x

* 30;

Steps:

x is float

30 is integer

Convert $30 \rightarrow 30.0$

Perform multiplication