Module 4: Semantic Analysis - Understanding Program Meaning

Imagine you're building a robot. The first step (lexical analysis) is like teaching the robot to recognize individual sounds or words. The second step (syntax analysis/parsing) is like teaching it to understand if those words are put together in a grammatically correct sentence. But even a grammatically perfect sentence can be nonsensical ("The square circle ran quickly").

Semantic Analysis is the third crucial step. It's where the compiler tries to understand the *meaning* of your program. It asks questions like: "Does this operation make sense with these types of data?", "Did the programmer declare this variable before using it?", or "Is this function call using the right number and types of arguments?" If something doesn't make logical sense according to the rules of the programming language, the semantic analyzer finds it and reports an error.

4.1 The Need for Semantic Analysis: Beyond Grammar

The parser (syntax analysis) ensures that your program follows the grammatical rules of the language. For example, it checks if every `if` has a corresponding `then` (or `else` block), or if parentheses are matched correctly. However, syntax alone isn't enough to guarantee a runnable, correct program. Semantic analysis steps in to check the "logic" and "meaning" of your code.

Why is it absolutely necessary? Let's break down the common types of semantic checks:

1. **Type Checking: Are You Mixing Apples and Oranges?**
    - **What it is:** This is the most common and fundamental semantic check. Every piece of data in a program has a "type" (like integer, floating-point number, text string, boolean true/false). Type checking makes sure that you're only performing operations that are *sensible* for those types.
    - **Analogy:** You can add two numbers (e.g., 5 + 3). You can also combine two words to make a phrase ("hello" + "world"). But can you add a number and a word (e.g., 5 + "hello")? In most programming languages, no. That's a type error.
    - **Examples of what it catches:**
        - `int age = "twenty";` (Trying to put text into a whole number variable.)
        - `bool isActive = 10 / 2;` (Trying to put the result of a division, which is a number, into a true/false variable.)
        - `"apple" - "pie";` (Trying to subtract text strings, which is usually not allowed.)
    - **Why it's important:** Catches common programming mistakes early, *before* your program even runs. This prevents frustrating "runtime errors" that are harder to debug. It also enforces the strictness (or flexibility) of a language's type system, making your code more predictable.
2. **Undeclared Variables and Functions: Have You Introduced Yourself?**

- **What it is:** Before you use a variable or call a function in your program, you usually have to "declare" it. This tells the compiler what it is (e.g., "I'm going to use a variable called 'myCount' which will hold an integer"). Semantic analysis checks if every name you use has been properly introduced.
- **Analogy:** Imagine trying to talk about a person named "Zephyr" in a conversation without ever explaining who Zephyr is. Your listener would be confused. The compiler gets confused too!
- **Examples of what it catches:**
  - `myVariable = 10;` (If `myVariable` was never declared with `int myVariable;` or similar.)
  - `calculateSum(a, b);` (If `calculateSum` function was never defined.)
- **Why it's important:** Ensures that every name refers to a known entity, preventing typos or forgotten declarations from causing mysterious bugs.

3. **Ambiguous Overloading Resolution: Which One Did You Mean?**
   - **What it is:** Some programming languages allow "overloading." This means you can have multiple functions with the *same name*, as long as they take different types or numbers of inputs (parameters). Similarly, operators like `+` might do different things (add numbers, combine strings). Semantic analysis figures out which specific version of the function or operator you intend to use based on the context (the types of data you provide).
   - **Analogy:** Imagine having a command "open." You might say "open the door" (meaning physically unlatch it) or "open the file" (meaning load it into memory). The word "open" is overloaded, and the context tells you which action is intended.
   - **Examples:**
     - You might have `print(int num)` and `print(string text)`. If you call `print(42);`, the semantic analyzer knows to use the integer version. If you call `print("Hello");`, it picks the string version.
     - `a + b;` could mean integer addition if `a` and `b` are integers, or floating-point addition if they are floats, or even string concatenation if they are strings.
   - **Why it's important:** Makes the language more flexible and natural, allowing programmers to use intuitive names or symbols for related operations without ambiguity. The compiler handles the underlying complexity.
4. **Access Control (Scope Checking): Who's Allowed to See This?**
   - **What it is:** Programming languages have rules about where variables and functions can be seen and used. This is called "scope." A variable declared inside a function is usually only visible *within* that function (local scope). A variable declared outside all functions might be visible everywhere (global scope). Object-oriented languages also have access modifiers like "public" or "private." Semantic analysis enforces these visibility rules.
   - **Analogy:** A secret message intended only for people inside a specific room. Someone outside that room shouldn't be able to read it.
   - **Examples of what it catches:**
     - Trying to use a variable `x` that was declared inside `functionA` from `functionB`.
     - Trying to access a `private` member of a class directly from outside that class.

    - **Why it's important:** Helps organize code, prevents unintended modifications of data, and supports modular programming practices like encapsulation.
5. **Return Type Checking: Did You Deliver What You Promised?**
    - **What it is:** When you define a function, you often state what type of value it will "return" (e.g., `int calculate_sum(...)` means it will give back an integer). Semantic analysis verifies that the function actually returns a value of that type, and that all possible paths through the function lead to a return statement (if a return value is expected). It also checks that functions declared as `void` (meaning they return nothing) don't try to return a value.
    - **Analogy:** If you promise to bring back an apple from the store, you shouldn't come back with a banana, or nothing at all!
    - **Examples of what it catches:**
        - A function declared to return an `int` actually returns a `string` in some cases.
        - A function declared `void` (no return value) has a `return 5;` statement.
    - **Why it's important:** Ensures consistency and correctness in how functions interact, preventing subtle bugs that might only appear during specific execution paths.
6. **Control Flow Statement Validation:**
    - **What it is:** Keywords like `break` and `continue` are special. They alter the normal flow of execution within loops or switch statements. Semantic analysis ensures they are only used in valid contexts.
    - **Analogy:** You can only "take a shortcut" or "skip ahead" if you're actually *inside* a race or a defined path. You can't just `break` out of thin air.
    - **Examples of what it catches:**
        - `break;` (if it's just floating in the code, not inside a `for`, `while`, or `switch`.)
    - **Why it's important:** Prevents nonsensical jumps in program execution, ensuring the control flow logic is sound.

The Indispensable Symbol Table:

To perform all these checks, the semantic analyzer relies heavily on a data structure called the **Symbol Table**. Think of the symbol table as the compiler's central database for all the names (identifiers) used in your program.

- **What it stores:** For each identifier, the symbol table keeps track of vital information discovered during earlier phases and updated by semantic analysis:
    - Its name (e.g., "myVariable").
    - Its **type** (e.g., `int`, `float`, `string`, `bool`).
    - Its **scope** (where in the program it's visible – global, local to a function, local to a block, etc.).
    - What kind of entity it is (variable, function, array, class, constant).
    - For functions: the number and types of its parameters, and its return type.
    - Any other relevant properties (e.g., whether it's read-only, its memory address once allocated).
- **How it's used:**

- When the semantic analyzer encounters an identifier, it looks it up in the symbol table to retrieve its stored information.
- If the identifier isn't found, it's an "undeclared" error.
- If it is found, the analyzer uses the type and scope information to perform checks (e.g., "Can I add an `int` to a `string`? No.").
- When new declarations are processed, the symbol table is updated with the new identifier's details.

4.2 Abstract Syntax Trees (ASTs): The Meaningful Blueprint

After the parser checks for grammatical correctness, it usually creates a **Parse Tree** (or concrete syntax tree). This tree is a very detailed representation of how the program matches the grammar rules, often including many keywords and intermediate grammatical steps that are not directly about the program's *meaning*.

For semantic analysis and later stages, we need a simpler, more concise representation that focuses on the core structure and meaning. This is where the **Abstract Syntax Tree (AST)** comes in.

What is an AST? Think of it as a blueprint:

An AST is a simplified, cleaned-up version of the parse tree. It throws away all the "syntactic noise" (like extra parentheses, semicolons, or redundant keywords from the grammar) and focuses only on the essential elements that define the program's structure and operations.

Key Characteristics that make ASTs perfect for Semantic Analysis:

- **Only Meaningful Elements:** Each node in an AST represents a significant construct in the programming language – like an operation, a variable, a literal value, an assignment, a loop, or a function call. It doesn't have nodes for intermediate grammatical rules.
- **Hierarchical Relationship:** The parent-child relationships in an AST show how different parts of the program relate to each other logically. For example, an "addition" node would have its two "operands" as children.
- **More Abstract, Less Concrete:** It's "abstract" because it doesn't show the exact syntax used; it shows the *idea* of the operation.
- **Directly Usable:** An AST is much easier to work with for semantic checks and code generation because you're dealing directly with concepts like "add," "assign," "if-then-else" rather than complex grammatical derivations.

Let's visualize ASTs for common programming constructs:

1. **Expressions: Capturing Calculation Order**
    - **Source Code:** `result = a + b * c;`
    - **The Problem with a Parse Tree:** A parse tree would show many intermediate steps for operator precedence (like `Expression -> Term -> Factor`). It would be bulky.
    - **AST for `a + b * c`:**

```
  +
 / \
a   *
   / \
  b   c
```

- **Explanation:** This AST immediately tells you that `b` is multiplied by `c` first, and then that result is added to `a`. The operators (`+`, `*`) become internal nodes, and the variables (`a`, `b`, `c`) become leaf nodes. This structure directly reflects the order of operations.

2. **Assignment Statements: Storing Values**
   - **Source Code:** `total = initial_value + 10;`
   - **AST:**

```
     =
    / \
 total  +
       / \
initial_value 10
```

- **Explanation:** An assignment statement is typically represented by an assignment node (`=`), with two children: the variable on the left-hand side (where the value goes) and the expression on the right-hand side (the value itself).

3. **Control Flow Statements: Guiding Program Path**
   - **If-Else Statement: Making Decisions**
     - **Source Code:**

```
if (score > 90) {
    grade = 'A';
} else {
    grade = 'B';
}
```

-
-
- **AST:**

```
      IfElse
     /  |   \
    >   =    =
   / \ / \  / \
```

```
score 90 grade 'A' grade 'B'
```

- **Explanation:** An `IfElse` node has three main branches:
  1. The **condition** (e.g., `score > 90`).
  2. The **"then" part** (the code to execute if the condition is true).
  3. The **"else" part** (the code to execute if the condition is false).
- Notice how the inner assignment statements also become subtrees.

- **While Loop: Repeating Actions**
  - **Source Code:**

```
while (count < 5) {
    count = count + 1;
}
```

- **AST:**

```
    While
    /    \
   <      =
  / \    / \
count 5  count  +
           / \
         count 1
```

- **Explanation:** A `While` loop node has two main branches:
  1. The **loop condition** (e.g., `count < 5`).
  2. The **loop body** (the statements inside the loop).

- **Function Call:**
  - **Source Code:** `print_message("Hello", count);`
  - **AST:**

```
 Call
 /   \
```

- print_message Arguments
  /
  "Hello" count
  ```

  * Explanation: A Call node represents a function invocation. Its children typically include the function name and a list of its arguments.

**How ASTs are used in Semantic Analysis:**

The AST becomes the central data structure that the semantic analyzer traverses.

1. **Walk the Tree:** Semantic analysis often involves one or more "walks" (traversals) of the AST, typically a depth-first traversal (visiting children before processing the parent, or vice versa, depending on the attribute type).
2. **Gather Information:** As the analyzer visits each node, it gathers information. For example, when it visits an expression node, it might calculate and determine the *type* of that expression.
3. **Update Symbol Table:** When it encounters a declaration node (like `int x;`), it adds `x` to the symbol table with its type (`int`) and scope information. When it encounters a usage of `x`, it looks it up in the symbol table to retrieve its type for type checking.
4. **Annotate AST Nodes:** The computed attributes (like the type of an expression, or a pointer to its symbol table entry) are often *stored directly on the AST nodes* themselves. This enriches the AST, making it even more useful for later compilation stages.
5. **Detect Errors:** If a semantic rule is violated (e.g., trying to add a string and an integer), the analyzer reports an error message, often pointing to the specific line or node in the source code.

By creating and working with ASTs, the compiler moves from simply knowing *how* the code is written (syntax) to understanding *what* the code intends to do (meaning).

## 4.3 Attribute Evaluation and Syntax-Directed Translation Schemes (STDS): The Rules of Meaning

Now that we have the AST, how does the semantic analyzer actually *do* its work of checking meaning and gathering information? It uses a powerful concept called **Attribute Evaluation**, guided by rules defined in **Syntax-Directed Translation Schemes (STDS)**.

**Attribute Evaluation: Attaching Information to Your Blueprint**

- **What are Attributes?** Think of attributes as sticky notes you attach to the nodes of your AST (or parse tree). Each sticky note holds a piece of information that is important for understanding the meaning of that part of the program.
  1. **Example:** For an expression node like `a + b`, you might attach an attribute called `type` that holds the resulting data type (e.g., `int` or `float`). For a variable node like `x`, you might attach an attribute called `symbolTableEntry` that points to its entry in the symbol table, where all its details (type, scope) are stored.
- **Two Flavors of Attributes (Information Flow):**
  ```

1. **Synthesized Attributes (Information Flows UP the tree):**
    - **Idea:** The value of a synthesized attribute at a node is calculated from the attribute values of its *children* nodes. Information moves from the bottom of the tree upwards towards the root.
    - **Analogy:** Imagine calculating the total cost of a shopping cart. You need the cost of each individual item (children nodes), and then you sum them up to get the total cost for the cart (parent node). The information (individual costs) is synthesized up to the total.
    - **Common Use Cases:**
        - **Type of an expression:** The type of `a + b` depends on the types of `a` and `b`.
        - **Value of a constant expression:** The value of `5 * (2 + 3)` is synthesized from the values of its sub-expressions.
        - **Size/offset of a data structure:** The size of a record/struct is the sum of the sizes of its members.
    - **Evaluation Order:** Typically evaluated using a **post-order traversal** (depth-first traversal where you process children first, then the parent).
2. **Inherited Attributes (Information Flows DOWN or ACROSS the tree):**
    - **Idea:** The value of an inherited attribute at a node is calculated from the attribute values of its *parent* node or its *siblings*. Information moves from the top down or horizontally.
    - **Analogy:** Imagine a project manager (parent node) assigning a "deadline" (inherited attribute) to different tasks (child nodes). Or, a variable's declaration might tell its usage nodes what its type is.
    - **Common Use Cases:**
        - **Expected type:** An assignment statement `x = expression;` might pass the *expected type* of `x` down to the `expression` node so that the expression can be checked for type compatibility or implicitly converted.
        - **Scope information:** A block of code (`{ ... }`) might pass down its scope context to declarations within it.
        - **Contextual information:** Information from the left sibling might influence a right sibling (e.g., in a list of declarations, the type declared for the first variable might be inherited by the subsequent ones if not explicitly specified).
    - **Evaluation Order:** Often evaluated using a **pre-order traversal** (depth-first traversal where you process the parent first, then its children).

Syntax-Directed Translation Schemes (STDS): The Recipe for Attributes

- **Concept:** An STDS is a formal way to specify how attributes are computed and how semantic actions are performed by associating them directly with the grammar rules (productions). It's essentially a set of "recipes" for how to build or decorate your AST based on the grammatical structure.

- **Structure:** Each rule in an STDS looks like a grammar production with embedded "semantic actions" (blocks of code) that explain what to do when that rule is applied during parsing or tree traversal.
  `A -> α { semantic_action }`
  - `A -> α`: This is a regular grammar production (e.g., `Expression -> Expression + Term`).
  - `{ semantic_action }`: This is a piece of code (often written in the implementation language of the compiler, like C or Java) that gets executed when the parser recognizes this specific grammatical pattern.
- **What Semantic Actions Do:**
  - **Attribute Calculation:** The primary purpose. They read attribute values from children/parent/siblings and compute new attribute values for the current node.
  - **Symbol Table Operations:** Add new entries to the symbol table for declarations, or look up entries for variable/function usage.
  - **Error Reporting:** If a semantic rule is violated during attribute computation (e.g., type mismatch), the action reports an error.
  - **Intermediate Code Generation:** While often a separate phase, sometimes simple intermediate code (like three-address code) can be generated during semantic analysis as attributes.

Detailed Example: Type Checking an Addition with STDS

Let's use a very simplified grammar and imagine how semantic rules would work.

Grammar Rules:

1. `E -> E1 + T` (An Expression is an Expression plus a Term)
2. `E -> T` (An Expression can just be a Term)
3. `T -> num` (A Term can be a number literal)

Attributes: We want to compute a synthesized attribute `type` for each `E` and `T` node.

STDS (Conceptual):

```
// Rule 1: E -> E1 + T
E.type = CheckAddType(E1.type, T.type); // CheckAddType is a helper function

// Rule 2: E -> T
E.type = T.type;

// Rule 3: T -> num
T.type = INT_TYPE; // Assume all number literals are integers for simplicity
```

Now, let's trace `x + y` where `x` is `int` and `y` is `float` (assume `x` and `y` are simple numbers for this trace):

AST:

```
    E (result_type)
   /|\
  E1 + T
 /     \
x       y
```

Attribute Evaluation Trace (Post-order traversal for synthesized attributes):

1. **Visit `x` (leaf node):**
    - This node corresponds to `T -> num`.
    - Action: `T.type = INT_TYPE`. So, `x.type` becomes `INT_TYPE`.
2. **Visit `y` (leaf node):**
    - This node corresponds to `T -> num`.
    - Action: `T.type = FLOAT_TYPE` (let's assume `y` was identified as a float literal). So, `y.type` becomes `FLOAT_TYPE`.
3. **Visit `+` node (parent of `x` and `y` in the `E1 + T` production):**
    - This node represents the `E -> E1 + T` production.
    - Action: `E.type = CheckAddType(E1.type, T.type)`.
    - Here, `E1.type` is `x.type` (INT_TYPE) and `T.type` is `y.type` (FLOAT_TYPE).
    - `CheckAddType(INT_TYPE, FLOAT_TYPE)` would likely return `FLOAT_TYPE` (because integers are usually promoted to floats in mixed-type arithmetic to avoid losing precision).
    - So, the `type` attribute of the top `E` node (representing `x + y`) becomes `FLOAT_TYPE`.

What happens if there's a type error?

If `y` was a `STRING_TYPE` in the above example:

1. `x.type` = `INT_TYPE`.
2. `y.type` = `STRING_TYPE`.
3. When evaluating `E.type` for `x + y`:
    - `CheckAddType(INT_TYPE, STRING_TYPE)` would detect an incompatible operation.
    - It would then:
        - Report an error message: "Error: Cannot add an integer and a string."
        - Set `E.type = ERROR_TYPE` (a special type to propagate the error, preventing further meaningless checks on this expression).

Tools that use STDS Concepts:

Compiler-compiler tools like **Yacc (Yet Another Compiler Compiler)** or **Bison** (GNU version of Yacc), which are typically used for parser generation, also allow you to embed semantic actions directly into your grammar rules. This means that as the parser builds the

AST, it can simultaneously execute these semantic actions to perform attribute evaluation and semantic checks.

In summary, semantic analysis is a critical guardian of program correctness, ensuring that your code is not just grammatically sound but also logically meaningful. It leverages ASTs as its primary data structure and employs attribute evaluation and syntax-directed translation schemes to systematically check and enrich the program's representation before it moves on to code generation.