Semantic Analysis

What is Semantic Analysis?

Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination. Now we'll move forward to semantic analysis, where we delve even deeper to check whether they form a sensible set of instructions in the programming language. For a program to be semantically valid, all variables, functions, classes, etc. must be properly defined, expressions and variables must be used in ways that respect the type system, access control must be respected, and so forth.

Semantic analysis is a pass by a compiler that adds semantic information to the parse tree and performs certain checks based on this information. It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which executable code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.) Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding). Sometimes also some early code optimization is done in this phase.

A large part of semantic analysis consists of tracking variable/function/type declarations and type checking. In many languages, identifiers have to be declared before they're used. As the compiler encounters a new declaration, it records the type information assigned to that identifier. Then, as it continues examining the rest of the program, it verifies that the type of an identifier is respected in terms of the operations being performed.
For example, the **type** of the **right side** expression of an assignment statement should **match** the **type** of the **left side**, and the left side needs to be a properly declared and assignable identifier. The **parameters of a function** should **match** the **arguments** of a function call in both number and type. The language may require that identifiers be unique, thereby forbidding two global declarations from sharing the same name. Arithmetic operands will need to be of numeric—perhaps even the exact same type (no automatic int-to-double conversion, for instance). **These are examples of the things checked in the semantic analysis phase.**
Some semantic analysis might be done right in the middle of parsing. As a particular construct is recognized, say an addition expression, the parser action could check the two operands and verify they are of numeric type and compatible for this operation.

Types and Declarations

We begin with some basic definitions to set the stage for performing semantic analysis.
A **type** is a set of values and a set of operations operating on those values. There are **three categories of types in most programming languages**:

Base types: **int**, **float**, **double**, **char**, **bool**, etc. These are the primitive types provided directly by the underlying hardware. There may be a facility for user-defined variants on the base types.
Compound types: **arrays, pointers, records, struct**, **union, classes,** and so on. These types are constructed as aggregations of the base types and simple compound types.
Complex types: **lists, stacks, queues, trees, heaps, tables**, etc. You may recognize these as abstract data types. A language may or may not have support for these sorts of higher-level abstractions.
In many languages, a programmer must first establish the name and type of any data object (e.g., variable, function, type, etc). In addition, the programmer usually defines the lifetime. A **declaration** is a statement in a program that communicates this information to the compiler. The basic declaration is just a name and type, but in many languages it may include modifiers that control visibility and lifetime (i.e., **static** in C, **private** in Java). **Some languages also allow declarations to initialize variables, such as in C**, where you can declare and initialize in one statement. The following C statements show some example declarations:

```
double calculate(int a, double b); // function prototype

int x = 0;   // global variables available throughout
double y;    // the program
int main()
{
 int m[3];  // local variables available only in main
 char *n;
 ...
}
```

Function declarations or prototypes serve a similar purpose for functions that variable declarations do for variables. Function and method identifiers also have a type, and the compiler can use ensure that a program is calling a function/method correctly. **The compiler uses the prototype to check the number and types of**

arguments in function calls. The location and qualifiers establish the visibility of the function (Is the function **global**? **Local** to the module? **Nested** in another procedure? Attached to a class?) Type declarations (e.g., C typedef, C++ classes) have similar behaviors with respect to declaration and use of the new typename.

<u>Type Checking</u>

Type checking is the process of verifying that each operation executed in a program respects the type system of the language. **This generally means that all operands in any expression are of appropriate types and number**. Much of what we do in the **semantic analysis phase is type checking**. Sometimes the rules regarding operations are defined by other parts of the code (as in function prototypes), and sometimes such rules are a part of the definition of the language itself (as in "both operands of a binary arithmetic operation must be of the same type"). If a problem is found, e.g., one tries to add a char pointer to a double in C, we encounter a **type error**. A language is considered **strongly-typed** if each and every type error is detected during compilation. Type checking can be done compilation, during execution, or divided across both.

Static type checking is done at **compile-time**. The information the **type checker** needs is obtained **via declarations and stored in a master symbol table**. After this information is collected, the types involved in each operation are checked. It is **very difficult for a language that only does static type checking** to meet the full definition of strongly typed. Even motherly old Pascal, which would appear to be so because of its use of declarations and strict type rules, cannot find every type error at compile time. This is because many type errors can sneak through the type checker.

For example, if a and b are of type int and we assign very large values to them, **a * b** may not be in the acceptable range of ints, or an attempt to **compute the ratio between two integers may raise a division by zero**. **These kinds of type errors** usually **cannot be detected at compile time**. **C** makes a somewhat paltry attempt at strong type checking—things as the lack of array bounds checking, no enforcement of variable initialization or function return create loopholes.

Dynamic type checking is implemented by **including type information** for each data location at **runtime**.
For example, a variable of **type double** would contain both the **actual double** value and some kind of tag indicating **"double type"**. The execution of any operation begins by first checking these **type tags**. The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts execution.

For example, when an **add operation** is invoked, it **first** examines the **type tags** of the **two operands** to ensure they are **compatible**. LISP is an example of a language that relies on **dynamic type checking**. Because LISP **does not** require the programmer to **state the types of variables at compile time**, the compiler cannot perform any analysis to determine if the type system is being violated. But **the runtime type system** takes over during execution and ensures that type integrity is maintained. **Dynamic type checking** clearly comes with a runtime performance penalty, **but it usually much more difficult** to subvert and can report errors that are not possible to detect at compile-time.