50006 - Compilers - (Dr Dulay) Lecture 4

Oliver Killane

10/04/22

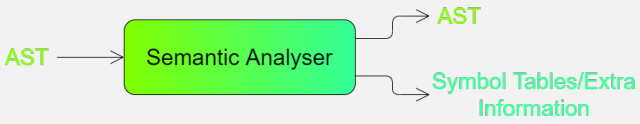
Semantic Analysis Overview

Lecture Recording

Lecture recording is available here

Definition: Semantic Analysis/Context Analysis

Compiler phase which checks (statically at compile time) if the program is semantically valid within the rules of the language. It also provides information for the next phases (code generation) such as creating symbol tables of identifiers and their types.



Some languages perform very few checks (e.g Python is highly dynamic, e.g types checked at runtime), to many (e.g Rust - among the most extreme) .

Semantic analysers can be generated from **attribute grammars** (which associate semantic rules with AST node types), however they are typically hand-written.

Typical Semantic Checks

variable Declaration

$$< Type >=< Id >$$

- **Check the type is in scope (the type is declared)**
  e.g $someclass = somevar$ the $someclass$ must be defined, if imported from another module/file further checks for qualified names, overloading etc must be done.
- **Check a declaration of the type is valid**
  The language may prevent some declarations, e.g $void\ a$;, or where the type is an abstract class.

  Furthermore some langagues place constraints on the order of declarations (e.g forward declaration in C).
- **Check the idenfitier in the current scope**
  If the language does not allow variable shadowing (renaming in inner scopes).
- **and more...**

Assignment

$$< Id >=< Expr >$$

- **Check the identifier is valid**
  Includes checking the scope to ensure it has been declared, as well as rules on variable types (e.g parameters).
- **Check the expression**
  The inner expression must be semantically analysed using rules for expressions.

- **Check the types**
  Given the expression is valid, it will have a type (or range of possible types). The semantic analyser must check it can coalesce this type into that of the assignment.

  It may also need to check the ranges, e.g assigning a constant larger than 255 to an unsigned byte-size type (e.g $u8$ in Rust or *unsigned char* in C).

Array Declaration

$$< Type > < Id > [< Size >] \dots$$

- **Check the declared Type is valid**
  Normal checks for if the type is valid, declared/in-scope. It may not be valid to create arrays of a given type (e.g $voidarray[3]$ is invalid, though $void * array$ is valid in C)
- **Check the size is valid**
  Type check the size expression, potentially only allowing constants (e.g for stack allocated arrays a size may be required).
- **Checking for scoping**
  Check for rules including variable shadowing, just as with declarations.
- **Array Size Warnings**
  Some array sizes may be too large (e.g large stack allocated arrays risking stack overflow in a recursive function), so a warning may need to be created.

Array Element Assignment

$$< Id > [< Index >] \cdots =< Expr >$$

- **Variable assignment scope checks**
  To check if the variable identifier is in scope.
- **Index and Type Check**
  We must check that the idenfifer's type allows it to be indexable (either it is an array or potentially implements some interface allowing indexing).

  The type of the assigned expression must match the de-indexed type of the variable (e.g $int\ a[][]; a[3] = \{1, 2, 3\}$ is correct as $a[i]$ is an integer array).
- **Bounds Checks**
  For each index we may need to check bounds (e.g index is within the bounds)

Function Declarations

$$< Ret\ Type > < Name > (< Param\ Type > < Param\ Id >, \dots)\{\dots\}$$

- **Checking types of return and the parameters**
  Normal variable declaration checking.
- **Checking the function name declared**
  The language may allow for function overloading, or defining functions in nested scopes (modules).
- **Checking scoping rules**
  Some languages may have rules for parameter scopes, for example in WACC the parameters exist in a scope enclosing the function body.

- **Checking for returns**
  Some languages require the return type of the body to be correct (and have no incorrect implicit returns - i.e path through function can end without a return).

Function Call

$$< Fun\ Name > (< Expr >, \dots)$$

- **Checking arguments are parameter compatible**
  Check the expressions provided are valid expressions, and if so that they can be matched with the type of the parameters.
- **Checking the Return Type**
  The expression will evaluate to the type of the function call. Hence if there is no return type (e.g returns *void* in C) this must be considered (e.g would be fine in a call statement, but not as part of an integer expression).

Type Checking in Languages

Type systems highly vary between languages:

- **Static vs Dynamic Typing** If types are checked at compile time, vs at runtime.
- **Strong Vs Weak Typing** If types can be cast and coerced easily.
  For example Python is dynamically and strongly typed, so when type checking occurs (at runtime) types must be explicity converted.

  However C is statically weak, meaning types are checked at compile time, but can be easily cast and converted (e.g cast any pointer type to any other pointer type)
- **Type Inference**
  Some languages can use values provided, functions used, return types etc to infer the type of a variable at compile time without it being explicitly stated.

  An example of this is Rust which has one of the most powerful such systems.
- **Function Overloading** Multiple different functions with the same name in the same scope. If function signatures differ, the compiler can determine which function to use in calls.

  In statically typed languages this is simpler (as types are known), some dynamically typed languages also support this.
  Python allows overloading, but the number of arguments must differ (as the type system is not yet powerful enoguh to distinguish between identical signatures).

  Despite its powerful type system, Rust avoids this (a language design choice) as overloading is considered unhelpful & to potentially make code more difficult to discipher by programmers.
- **Polymorphic Typing** Some languages allow for types to be based on interfaces, parent classes, etc. Another form is generics.
- **Assignment-Compatability and Type-Equivalence** Differs between languages, rules concering how types are matched, and what types can be matched
- **Type Coercion/casting** Some languages allow implicit casting (e.g C) while others require explicit (e.g Rust, Python).

- **Primitive Sizes** Basic data types such as integers, floats are implemented differently between languages.
  For example some languages allow for arbitrary size integers as a language feature (e.g Python). This has negative performance implications, so high performance languiages such as Rust and C use fixed size integers (as represented in the architectures they compile to).

Symbol Tables

Lecture Recording

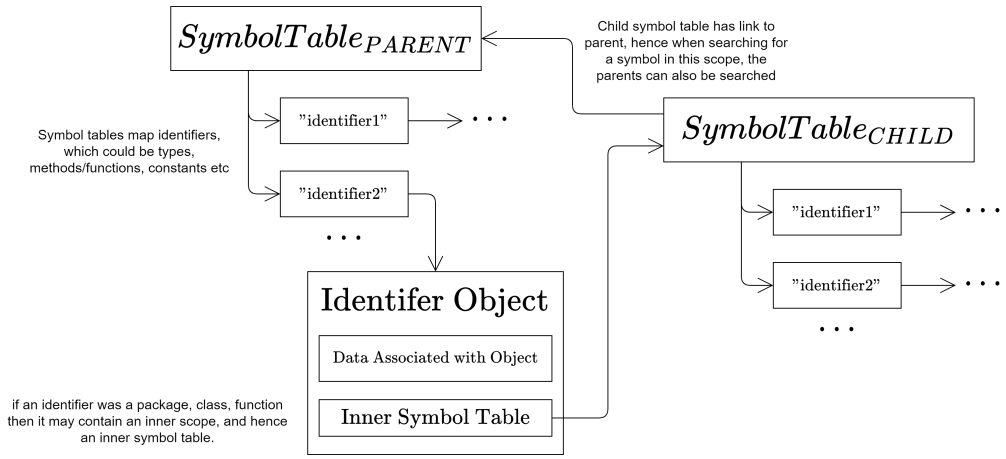
Lecture recording is available here

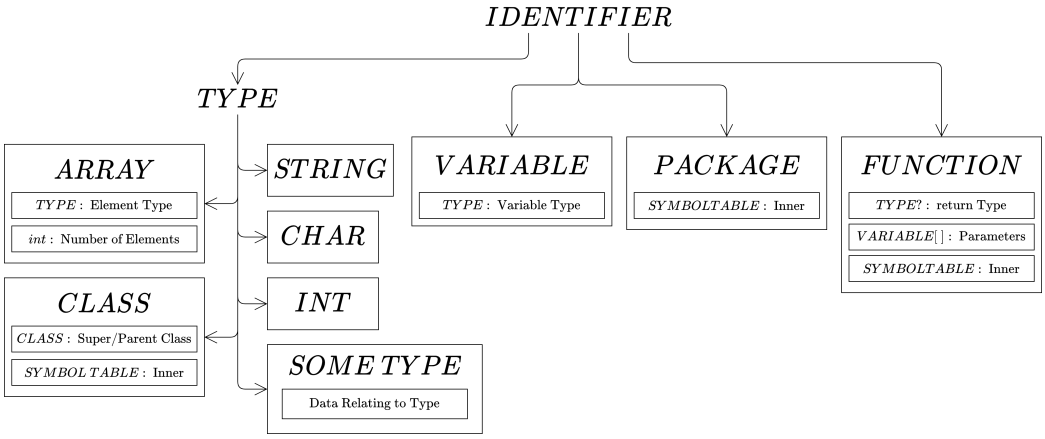
Definition: Symbol Table/Identifier Table

Rather than store semantic information for identifiers in AST nodes (which would require lengthly lookup during code generation) a structure is used to identify identifies with semantic information.

- For scoping a tree of symbol tables can be used. Alternatively a flat symbol table can be generated with identifiers being translated to some non-shadowed form (no name conflicts).
- In many languages identifiers are declared before use. This allows a single pass to build the symbol table.
- Map data structures are used for the tables to reduce the time required for identifier lookup.

Example Symbol Table Implementation





Lecture Recording

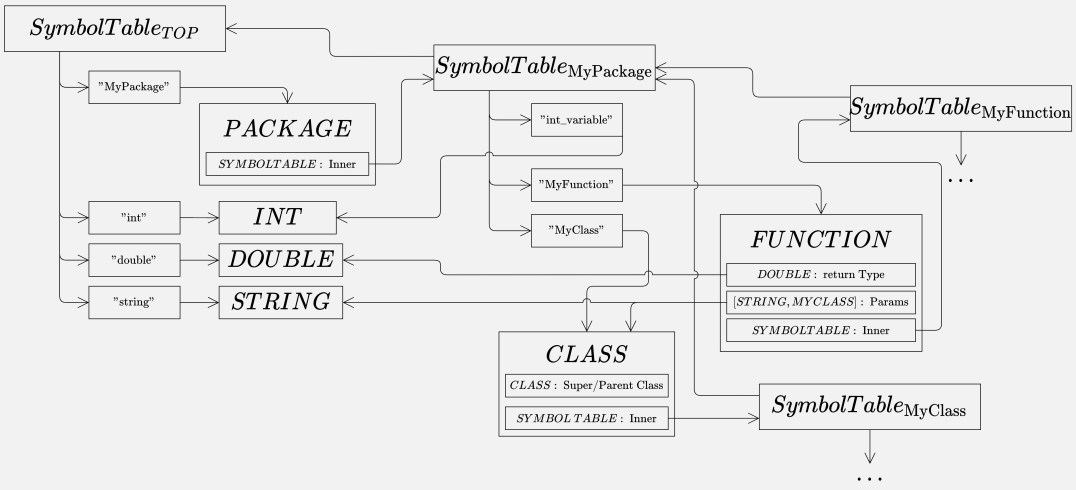
Lecture recording is available here

We can embed pointers the identifier objects inside the AST also.

When checking types, we can use ad-hoc rules for coercion (e.g can assign an *int* to a *double*). We may also want to use the class hierarchy (if no match, check if any super classes can match)

Example: Basic Package

```
1 package MyPackage {
2     int int_variable;
3     class MyClass { ... }
4     double MyFunction(String param_1, MyClass param_2) { ... }
5 }
```



We can then put our basic types, standard types, functions etc (default includes) in the top level symbol table.