CS143 Handout #21
Winter 2001

Semantic analysis

Handout written by Maggie Johnson and revised by me.

What is semantic analysis?

Parsing only verifies that the program consists of tokens arranged in a syntactically-valid combination, we now move on to *semantic analysis*, where we delve deeper to check whether they form a sensible set of instructions in the programming language. Whereas any old noun phrase followed by some verb phrase makes a syntactically-correct English sentence, a semantically-correct one has subject-verb agreement, gender is properly used, and the components go together to express an idea that "makes sense." For a program to be semantically correct, all variables, functions, classes, etc. are properly defined, expressions and variables are used in ways that respect the type system, access control isn't violated, and so on. Semantic analysis is the next-to-last phase of the front end and the compiler's last chance to weed out incorrect programs. We need to ensure the program is well-formed enough to continue on to the next phase where we generate code.

A large part of semantic analysis consists of tracking variable/function/type declarations and definitions and type checking. As we enter each new identifier in our symbol table, we need to record the type information of the declaration. Then, as we continue parsing the rest of the program, we make sure that the type of each identifier and expression 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 (i.e. not some sort of constant). The parameters of a function should match the arguments of a function call in both number and type. The language may require that identifiers are unique, disallowing a global variable and function of the same name. The operands to multiplication operation will need to be of numeric type, perhaps even the exact same type depending on the strictness of the language. These are examples of the things we check in the semantic analysis phase.

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 more or less directly by the machine upon which the compiler is implemented. There may be a facility for user-defined variants on the base types (such as C enums).
Compound types	arrays, pointers, records/structs/unions, classes, and so on. These are types constructed from aggregations of the base types.
Complex types	lists, stacks, queues, trees, heaps, tables, etc. You may recognize these as ADTs. A language may or may not have support for these sort of higher-level abstractions.

In many languages, a programmer must first establish the name and type of any data object (variable, function, type, etc). In addition, the programmer usually defines the lifetime. A *declaration* is a statement in a program that communicates to the compiler this information. The declaration indicates the name and type, while the placement of the declaration communicates the lifetime. Some languages also allow declarations to also 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, it establishes the type for that identifier so that later usage of that identifier can be validated for type-correctness. The compiler uses the prototype to check the number and types of arguments in function calls. The location and qualifiers on the prototype establish the visibility of the function— is the function global, local to the module, nested in another procedure, attached to a class, and so on. Type declarations (i.e. C typedef, C++ classes) have similar behaviors with respect to declaration and use of the new typename.

Type checking

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 no type error goes undetected, *weakly-typed* if there are various loopholes by which type errors can slip through. Type checking can be done when the program is compiled, when it is executed, or a combination of both.

Static type checking is done at compile-time. The information the type checker needs is obtained via declarations and stored in the 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 fall out of the acceptable range of ints, or an attempt to compute the ratio between two integers may raise a division by zero. These kind of type errors are cannot be detected at compile-time. Although C makes an attempt at strong type checking, it allows the type system to be comprised via the typecast operation. By taking the address of a location, casting to something inappropriate, dereferencing and assigning, you can wreak havoc on the type rules. The typecast basically suspends type checking, which, in general, is a pretty dangerous thing to do.

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.

Many compilers have built-in functionality for correcting some type errors that can occur. *Implicit type conversion*, or *coercion*, is when a compiler finds a type error and then changes the type of the variable to an appropriate type. This happens in C, for example, when an addition operation is performed on a mix of integer and floating point values. The integer values are implicitly promoted before the addition is performed. In fact, any time a type error occurs in C, the compiler searches for an appropriate conversion operation to insert into the compiled code to fix the error. Only if no conversion can be done, does a type error occur. In a language like C++, the space of possible automatic conversions can be enormous, which makes the compiler run more slowly and sometimes gives surprising results.

> The base types of FORTRAN consisted of just integers and reals. Thus, type checking was very simple in this language. Backus in 1981 recalled: “I think just because we did not like the rules as to what happened with mixed mode expressions, we decided ‘Let’s throw it out - it’s easier!’

Other languages are much stricter about type-coercion. Ada and Pascal, for example, provide almost no automatic coercions, requiring the programmer to take explicit actions to convert between various numeric types. The question of whether to provide a coercion capability or not is controversial. Coercions can free a programmer from worrying about details, but they can also hide serious errors that might otherwise have popped up during compilation. PL/I compilers are especially notorious for taking a minor error like a misspelled name and re-interpreting it in some unintended way. Here’s a particular classic example:

```
DECLARE (A, B, C) CHAR(3);
B = "123"; C = "456"; A = B + C;
```

The above PL/1 code declares A, B, and C each as 3-character array/strings. It assigns B and C string values and then adds them together. Wow, does that work? Sure, PL/I automatically coerces strings to numbers in an arithmetic context, so it turns B and C into 123 and 456, then it adds them to get 579. How about trying to assign a number to a string? Sure, why not! It will convert a number to string if needed. However, herein lies the rub: it converts the string using a default width of 8, so it actually converted the result to "   579". And because A was declared to only hold 3 characters, it will truncate (silently), and A gets assigned "   ". Probably not what you wanted, eh?

> The first design principle for PL/I was “Anything goes! If a particular combination of symbols has a reasonably sensible meaning, that meaning will be made official.”

Case study: ML data types

ML, or Meta-Language, is an important functional language developed in Edinburgh in the 1970’s. It was developed to implement a theorem prover, but recently, it has gained popularity as a general-purpose language. ML deals with data types in a novel way. Rather than require declarations, ML works very hard to infer the data types of the arguments to functions. For example,

```
fun mult x = x * 10;
```

requires no type information because ML infers that x is an integer since it is being multiplied by an integer. The only time a programmer must supply a declaration is if ML cannot infer the types. For example,

```
fun sqr x = x * x;
```

would result in a type error because the multiplication operator is *overloaded*, i.e., there exists a multiplication operation for reals and one for integers. ML cannot determine which one to use in this function, so the programmer would have to clarify:

```
fun sqr x:int = x * x;
```

The fact that types do not have to be declared unless necessary, makes it possible for ML to provide one of its most important features: *polymorphism*. A polymorphic function is one that takes parameters of different types on different activations. For example, a function that returns the number of elements in a list:

```
fun length(L) = if L = nil then 0 else length (tl(L)) + 1;
```

(Note: "tl" is a built-in function that returns all the elements after the first element of a list.) This function will work on a list of integers, reals, characters, strings, lists, etc. Polymorphism is an important feature of most object-oriented languages also. It introduces some interesting problems in semantic analysis, as we will see a bit later.

Designing a type checker

When designing a type checker for a compiler, here's the process to follow:

1. identify the types that are available in the language
2. identify the language constructs that have types associated with them
3. identify the semantic rules for the language

To make this process more concrete, we will present it in the context of Decaf. Decaf is a somewhat strongly-typed language like C since declarations of all variables are required at compile-time. In Decaf, we have base types (int, double, bool, string), and compound types (arrays and classes). An array can be made of any type (either a base type, a class, or out of other arrays). Classes are a bit special in that the class name must be declared before it can be used in a declaration. ADTs can be constructed using classes, but they aren't handled in any way differently than classes, so we don't need to consider them specially.

Now that we know the types in our language, we need to identify the language constructs that have types associated with them. In Decaf, here are the relevant language constructs:

constants	obviously, every constant has an associated type. A scanner tells us these types as well as the associated lexeme.
variables	all variables (global, local, and instance) must have a declared type of either int, double, bool, string, array, or class.
functions	functions have a return type, and each parameter in the function definition has a type, as does each argument in a function call.
expressions	an expression can be a constant, variable, function call, or some operator (binary or unary) applied to expressions. Each of the various expressions have a type based on the type of the constant, variable, return type of the function, or type of operands.

The other language constructs in Decaf (if, while, Print, assignments, etc.) also have types associated with them, because somewhere in each of these we find an expression.

The final requirement for designing a type checking system is listing the semantic rules that govern what types are allowable in the various language constructs. In Decaf, the operand to a unary minus must either be double or int, the expression used in a loop test must be of bool type, and so on. There are also general rules, not just a specific construct, such as all variables must be declared before use, all classes are global, and so on.

These three things together (the types, the relevant constructs, and the rules) define a *type system* for a language. Once we have a type system, we can implement a type checker as part of the semantic analysis phase in a compiler.

Type checker implementation

The first step in implementing a type checker for a compiler is to record type information into the symbol table record for each identifier. All a scanner knows is the name of the identifier so that it what is passed to the parser. Typically, the parser will create some sort of "declaration" record for all each identifier after parsing its declaration and store it in the symbol table, indexed by name. On encountering later uses of that identifier, the parser/semantic analyzer can lookup that name and find the matching declaration or report when no declaration has been found.

Let's consider an example. In Decaf, we have the following productions that are used to parse the declaration section of a function (where the user-defined variables are declared).

```
VariableDecl         ->    Variable ';'

Variable             ->    Type identifier

Type                 ->    int
                     ->    void
                     ->    bool
                     ->    double
                     ->    string
                     ->    class identifier
                     ->    Type '[' ']'
```

Consider the following variable declaration section:

```
int a;
double b;
```

The scanner stores the name for an identifier lexeme, which the parser records as an attribute attached to the token. When reducing the Variable production, we have the type associated with the "Type" symbol (passed up from the Type production) and the name associated with the "identifier" symbol (passed from the scanner). We add a new entry to our symbol table, declaring that identifier to be of that type.

Representing base types and array types are pretty straightforward. Function types can be represented as a collection of other types (the return type and list of formal parameter types). Classes are a bit more involved because the class needs to record a list of all fields (variables and methods) available in the class to enable access and type checking on the fields. Most likely this is done with some sort of symbol table. Classes also need to be able to support inheritance of all parent fields which might be implemented by linking the parent's symbol table into the child's or copying the symbol table of the parent to the child's.

Once we have the type information stored away and easily accessible, we then use this to check that the program follows the general semantic rules and the specific ones concerning the language constructs. For example, what types are allowable to be added? Consider Decaf's expression productions:

```
Expr          ->      Expr + Expr |
                      Expr - Expr |
                      Expr * Expr |
                      Expr / Expr |
                      .... |
                      LValue |
                      Constant

LValue        ->      OptReceiver identifier

Constant      ->      intConstant |
                      doubleConstant |
                      ...
```

In parsing an expression such as `x + 7`, we would apply the `LValue` and `Constant` productions to the two sides. Passed up from these would be either the declaration information for the variable or the type of the constant. When we are reducing the `Expr + Expr` production, we examine the type of each operand to determine if it is appropriate in this context, which in Decaf, means the two operands must be both int or both double.

The semantic analysis phase is all about verifying the language rules, especially those that are too complex or difficult to constrain in the grammar. To give you an idea, here are a few semantic rules from the Decaf spec

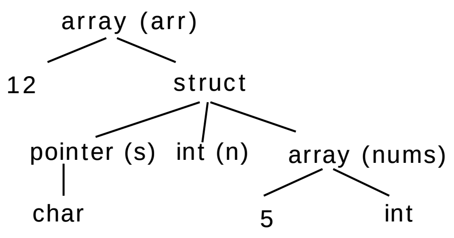
arrays:	• the index used in an array selection expression must be of integer type
expressions:	• the two operands to % must both be int. The result type is int. • `this` is bound to the receiving object within class scope, it is an error outside class scope
variables	• a variable declared of class type must refer to a defined class name
functions	• the type of each actual argument in a function call must be compatible with the formal parameter • if a function has a void return type, it may only use the empty return statement

As we can see from this list, much of the semantic checking has to do with types, but, for example, we also check that names are not re-used or that break only appears inside a loop. Thus, the implementation of a type checker consists of working through the defined semantic rules of a language and making sure those rules are applied consistently. We do this by placing actions in the grammar, so semantic analysis is performed as we parse.

Type equivalence of compound types

All the type checking we have studied thus far has pertained to base types. Many languages also require the ability to determine the equivalence of compound types. A common technique in dealing with compound types is to store the basic information defining the type in tree structures.

```
struct {
   char *s;
   int n;
   int nums[5];
} arr[12];
```



Here is a set of rules for building type trees:

arrays:	two subtrees, one for number of elements and one for the base type
structs:	one subtree for each field
pointers:	one subtree that is the type being referenced by the pointer

If we store the type information in this manner, checking the equivalence of two types turns out to be a simple recursive tree operation. Here's an outline of a recursive test for structural equivalence:

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2)
{
   if (tree1 == tree2)  // if same type pointer, must be equiv!
      return true;
   if (tree1->type != tree2->type)  // check types first
      return false;
   switch (tree1->type) {
      case T_INT: case T_DOUBLE: ...
         return true;              // same base type
      case T_PTR:
         return (AreEquivalent(tree1->child[0], tree2->child[0]);
      CASE T_ARRAY:
         return (AreEquivalent(tree1->child[0], tree2->child[0]) &&
                (AreEquivalent(tree1->child[1], tree2->child[1]);
      ...
```

The function looks simple enough, but is it guaranteed to terminate? What if the type being compared is a record that is recursive (i.e. contains a pointer to a record of the same type?) Hmmm… we need to be a bit more careful! To get around this, we could either mark the tree nodes as we traverse so we could detect a cycle or limit equivalence on pointer types to in name only.

User-defined types

The question of equivalence becomes more complicated when you add user-defined types. Many languages allow users to define their own types (e.g., using `typedef` in C, or `type` in Pascal). Here is a Pascal example:

```
type
   little = array[1..5] of integer;
   small = array[1..5] of integer;
   big = array[1..10] of integer;

var
   a, b: array[1..5] of integer;
   c: array[1..5] of integer;
   d, e: little;
   f, g: small;
   h, i: big;
```

When are two types the same? Which of the types are equivalent in the above example? It depends on how one defines "equivalence", the two main options are *named* versus *structured* equivalence. If the language supports named equivalence, two types are the same if and only if they have the same name. Thus `d` and `e` are type-equivalent, so are `f` and `g` and `h` and `i`. The variables `a` and `b` are also type-equivalent because they have identical (but unnamed) types. (Any variables declared in the same statement have the same type.) But `c` is a different, anonymous type. And even though the `small` type is a synonym for `little` which is a synonym for an array of 5 integers, Pascal, which only supports named equivalence, does not consider `d` to be type-equivalent to `a` or `f`. The more general form of equivalence is *structural equivalence.* Two types are structurally equivalent if a recursive traversal of the two type definition trees matches in entirety. Thus, the variables `a`-`g` are all structurally equivalent but are distinct from `h` and `i`.

Which definition of equivalence a language supports is a part of the definition of the language. This, of course, has an impact on the implementation of the type checker of the compiler for the language. Clearly, a language supporting named equivalence is much easier and quicker to type check than one supporting structural equivalence. But there is a trade-off. Named equivalence does not always reflect what is really being represented in a user-defined type. Which version of equivalence does C support? Do you know? How could you find out?

The first programming language that allowed compound and complex data structures was Algol 68. This language allowed for recursively defined type expressions, and it used structural equivalence.

Scope checking

One additional issue in semantic analysis is dealing with *scopes*. Many languages allow the declaration of global variables that are available anywhere in the program. Local variables, on the other hand, are only visible within certain sections. To understand how this is handled in a compiler, we need a few definitions. A *scope* is a section of program text enclosed by basic program delimiters, e.g., `{}` in C, or `begin-end` in Pascal. Many languages allow *nested scopes* that are scopes defined within other scopes. The scope defined by the innermost such unit is called the *current scope.* The scope defined by the current scope and by any enclosing program units are known as *open scopes*. Any other scope is a *closed scope.*

As we encounter identifiers in a program, we need to determine if the identifier is accessible at that point in the program. This is called *scope checking*. If we try to access a local variable declared in one function in another function, we should get an error message. This is because only variables declared in the current scope and in the open scopes containing the current scope are accessible.

An interesting situation can arise if a variable is declared in more than one open scope. Consider the following C program:

```
int a;
void Binky(int a) {
      int a;
      a = 2;
      ...
}
```

When we assign to `a`, should we use the global variable, the local variable, or the parameter? Normally it is the innermost declaration, the one nearest the reference, which wins out. Thus, the local variable is assigned the value 2. When a variable name is re-used like this, we say the innermost declaration *shadows* the outer one. Inside the `Binky` function, there is no way to access the other two `a` variables because the local variable is shadowing them.

There are two common approaches to the implementation of scope checking in a compiler. The first is to implement an individual symbol table for each scope. We organize all these symbol tables into a *scope stack* with one entry for each open scope. The innermost scope is stored at the top of the stack, the next containing scope is underneath it, etc. When a new scope is opened, a new symbol table is created and the variables declared in that scope are placed in the symbol table. We then push the symbol table on the stack. When a scope is closed, the top symbol table is popped. To find a name, we start at the top of the stack and work our way down until we find it. If we do not find it, the variable is not accessible and an error should be generated.

There is an important disadvantage to this approach, besides the obvious overhead of creating additional symbol tables and doing the stack processing. All global variables will be at the bottom of the stack, so the scope checking of a program with a lot of global variables and many levels of nesting can run slowly. The overhead of a table per scope can also contribute to memory bloat in the compiler.

The other approach to the implementation of scope checking is to have a single global table for all the scopes. We assign to each scope a scope number. Each entry in the symbol table is assigned the scope number of the scope it is contained in. A name may appear in the symbol table more than once as long as each repetition has a different scope number.

When we encounter a new scope during parsing, we increment a scope counter. All variables declared in this scope are placed in the symbol table and assigned this scope's number. If we then encounter a nested scope, the scope counter is incremented once again and any newly declared variables are assigned this new number. Using a hash table, new names are always entered at the front of the chains to simplify the searches. Thus, if we have the same name declared in different nested scopes, the first occurrence of the name on the chain is the one we want.

When a scope is closed, all entries with the closing scope number are deleted from the table. Any previously shadowed variables will now be accessible again. If we try to access a name in a closed scope, we will not find it in the symbol table causing an error to be generated. The disadvantage of the single combined symbol table is that closing a scope is an expensive operation since it requires traversing the entire symbol table.

There are two scoping rules that can be used in block-structured languages— *static scoping* and *dynamic scoping*. In static scoping, a function is called in the environment of its definition (i.e. its lexical placement in the source text), where in dynamic scoping, a function is called in the environment of its caller (i.e. using the runtime stack of function calls). For a language like C or Decaf,, the point is moot, because functions can only access their local variables or global ones, but not the local variables of any other functions. But other languages such as Pascal or LISP allow non-local access and thus need to establish scoping rules for this. If inside the Binky function, you access a non-local identifier x, does it consider the static structure of the program (i.e. the context in which Binky was defined, which may be nested inside other functions in those languages)? Or does it use the dynamic structure to examine the call stack to find the nearest caller who has such a named variable? What if there is no x in the enclosing context— can this be determined at compile-time for static scoping? What about dynamic? What kind of data structure are necessary at compile-time and run-time to support static or dynamic scoping? What can you do with static scoping that you can't with dynamic or vice versa? Over time, static scoping has won out over dynamic— what might be the reasoning it is preferred?

Object-oriented issues

Object-oriented languages present some interesting challenges in semantic analysis. The first issue concerns *sub-typing*. If a data type is part of a larger class, we say it is a subtype of that class. C's enums, for example, allow you to define new subtypes of int, similar to Pascal subrange type. A subtype is compatible with its parent type, which means an expression of the subtype can always be substituted where the general type was expected. This it will involve some additional work in type-checking to handle subtypes.

Inheritance can be thought of a particular form of subtyping. If a subclass has an "is-a" relationship with its parent, an instance of the subclass can be substituted wherever an instance of the superclass is needed. Some languages only support subtype relationships for inheritance (such as Java) which means all public features available in the superclass must still be public and available in a subclass. C++ gives control (via private inheritance) for a subclass to inherit implementation, but not interface, thus making the subclass not a subtype of its parent. Java interface support subtyping relationships among classes not related through inheritance.

In the usual form of inheritance, a derived class inherits both interface and implementation from the parent. Often a copy-based approach is used to implement inheritance. The storage for each derived class or subtype contains all the information needed from the parent because it was copied directly into the object.

Another issue in object-oriented languages is polymorphism. We saw this feature earlier when discussing ML, but the term takes on a slightly different meaning in a language like C++ or Java. Polymorphism in C++ refers to the situation in which objects belonging to different classes can respond to the same message. For example, if we have classes for boat, airplane, and car, the objects of these classes might all understand the message `go()`, but the action will be different for each class. In C++, polymorphism is implemented using *virtual functions*. Methods of the same name can have different definitions in different classes. For example, consider this excerpt from a hypothetical drawing program:

```
class Shape {
   public:
      virtual void Draw();
      virtual void Rotate(int degrees);
   ...
};

class Rect: public Shape {
   public:
      void Draw();
      void Rotate(int degrees) {}
   ...
};
class Oval: public Shape {
   public:
      void Draw();
      void Rotate(int degrees) {}
   ...
};
```

If we have an array of different shape objects, we can rotate all of them by placing the following statement inside a loop that traverses the array:

```
shapes[i].Rotate(45);
```

We are rotating all the objects in the list without providing any type information. The receiving object itself will respond with the correct implementation for its shape type.

The primary difference between virtual functions and non-virtual functions is their binding times. Binding means associating a name with a definition or storage location. In C++, the names of non-virtual functions are bound at compile time. The names of virtual functions are bound at run-time, at the time of the call to the method. Thus, the binding is determined by the class of the object at the time of the function call. To implement this, each virtual method in a derived class reserves a slot in the class definition record, which is created at run-time. A constructor fills in this slot with the location of the virtual function defined in the derived class, if one exists. If it does not exist, it fills in the location with the function from the base class.

Bibliography

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.

J. Tremblay, P. Sorenson, The Theory and Practice of Compiler Writing. New York, NY: McGraw-Hill, 1985.