Chapter 7 – Semantic Analysis

Now that we have completed construction of the AST, we are ready to begin analyzing the **semantics**, or the actual meaning of a program, and not simply its structure.

Type checking is a major component of semantic analysis. Broadly speaking, the type system of a programming language gives the programmer a way to make verifiable assertions that the compiler can check automatically. This allows for the detection of errors at compile-time, instead of at runtime.

Different programming languages have different approaches to type checking. Some languages (like C) have a rather weak type system, so it is possible to make serious errors if you are not careful. Other languages (like Ada) have very strong type systems, but this makes it more difficult to write a program that will compile at all!

To perform type checking, we must enter each variable in the program into a **symbol table**. This table is referenced throughout the semantic analysis stage whenever we need to evaluate the correctness of some code.

However, the mapping between variable names and their actual storage locations is not immediately obvious. A variable `x` in an expression could refer to a local variable, a function parameter, a global variable, or something else entirely. So, the first step of analysis is **name resolution**, in which we match up each variable use with the definition in the symbol table.

Once name resolution is completed, we have all the information necessary to check types. In this stage, we compute the type of complex expressions by combining the basic types of each value according to standard conversion rules. If a type is used in a way that is not permitted, the compiler will output an (ideally helpful) error that will assist the programmer in resolving the problem.

7.1 Overview of Type Systems

Most programming languages assign to every variable and value a **type**, which describes the interpretation of the data in that variable: is it an integer, a floating point number, a boolean, a string, a pointer, or something else. In many languages, these basic types can be combined into higher-order types such as enumerations, structures, and variant types to express complex constraints.

The type system of a language serves several purposes:

- **Correcness.** A compiler uses type information provided by the programmer to raise warnings or errors if a program attempts to do something improper. For example, it almost certainly an error to assign an integer value into a pointer variable, even though both might be implemented as a single word in memory. A good type system can help to eliminate runtime errors by flagging them at compile time instead.

- **Performance.** A compiler can use type information to find the most efficient implementation of a piece of code. For example, if the programmer tells the compiler that a given variable is a constant, then the same value can be loaded into a register and used many times, rather than constantly loading it from memory.

- **Expressiveness.** A program can be made more compact and expressive if the language allows the programmer to leave out facts that can be inferred from the type system. For example, in C-Minor , the `print` statement does not need to be told whether it is printing an integer, a string, or a boolean: the type is inferred from the expression and the value is automatically displayed in the proper way.

A programming language (and its type system) are commonly classified on the following axes:

- safe or unsafe
- static or dynamic
- explicit or implicit

In an **unsafe programming language**, it is possible to write valid programs that have wildly undefined behavior that violates the basic structure of the program. For example, in the C programming language, a program can construct an arbitrary pointer to modify any word in memory, and thereby change the data and code of the compiled program. Such power is probably necessary to implement low-level code like an operating system or a driver, but is problematic for general applications code.

For example, the following code in C is syntactically legal and will compile, but is unsafe because it writes data outside the bounds of the array `a[]`:

```
/* This is C code */
int i;
int a[10];
for(i=0;i<100;i++) a[i] = i;
```

In a **safe programming language**, it is not possible to write a program that violates its own definition. That is, no matter what input is given to a program written in a safe language, it will always execute in a well defined way that preserves the abstractions of the language. A safe programming language enforces the boundaries of arrays, the use of pointers, and the assignment of types to prevent undefined behavior. Most interpreted languages, like Perl, Python, and Java, are safe languages.

For example, in C#, the boundaries of arrays are checked at runtime, so that running off the end of an array has the predicable effect of throwing an `IndexOutOfRangeException`:

```
/* This is C-sharp code */
a = new int[10];
for(int i=0;i<100;i++) a[i] = i;
```

In a **statically typed language**, all typechecking is performed at compile-time, long before the program runs. This means that the program can be translated into basic machine code without retaining any of the type information, because all operations have been checked and determined to be safe. This yields the most high performance code, but does eliminate some kinds of convenient programming idioms.

Static typing is in most languages to distinguish between integer and floating point operations. While operations like addition and multiplication are usually represented by the same symbols in the source language, they are implemented with fundamentally machine code. For example, in the C language on an X86 machines, `(a+b)` would be translated to an `ADD.L` instruction for integers, but an `FMUL` instruction for floating point values.

In a **dynamically typed language**, type information is available at runtime, and stored in memory alongside the data that it describes. As the program executes, the safety of each operation is checked by comparing the types of each operand. If types are observed to be incompatible, then the program must halt with a runtime type error. This also allows for code that can explicitly examine the type of a variable. For example, the `instanceof` operator in Java allows one to test for types explicitly:

```
/* This is Java code */

public void sit( Furniture f ) {
   if (f instance of Chair) {
       System.out.println("Sit up straight!\n");
   } else if ( f instance of Couch ) {
       System.out.println("You may slouch.\n");
   } else {
       System.out.println("You may sit normally.\n");
   }
}
```

In an **explicitly typed language**, the programmer is responsible for indicating the types of variables and other items in the code explicitly. This requires more effort on the programmer's part, but reduces the possibility of unexpected errors. For example, in an explicitly typed language like C, the following code might result in an error or warning, due to the loss of precision when assigning a floating point to an integer:[1]

```
/* This is C code */
int x = 32.5;
```

Explicit typing can also be used to prevent assignment between variables that have the same underlying representation, but different meaning. For example, in C and C++, pointers to different types have the same implementation (a pointer) but it makes no sense to interchange them. The following could should generate an error or at least a warning:

```
/* This is C code */
int *i;
float *f = i;
```

In an **implicitly typed language**, the compiler will infer the type of variables and expressions (to the degree possible) without explicit input from the programmer. This allows for programs to be more compact, but can result in accidental behavior. For example, recent C++ standards now allow a variable to be declared with automatic type `auto`, like this:

```
/* This is C++11 code */
auto x = 32.5;
cout << x << endl;
```

The compiler determines that 32.5 has type `double`, and therefore `x` must also have type `double`. In a similar way, the output operator $<<$ is defined to have a certain behavior on integers, another behavior on strings, and so forth. In this case, the compiler already determined that the type of `x` is `double` and so it chooses the variant of $<<$ that operates on doubles.

[1] Not all C compilers will generate a warning, but they should!

7.2 The Symbol Table

The **symbol table** records all of the information that we need to know about every declared variable (and other named items, like functions) in the program. Each entry in the table is a `struct symbol` which is shown in Figure 7.1.

```
struct symbol {
    symbol_t kind;
    struct type *type;
    char *name;
    int which;
};

typedef enum {
    SYMBOL_LOCAL,
    SYMBOL_PARAM,
    SYMBOL_GLOBAL
} symbol_t;
```

Figure 7.1: The Symbol Structure

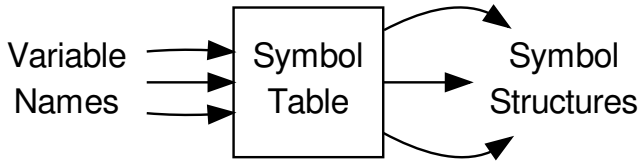
The `kind` field indicates whether the simply is a local variable, a global variable, or a function parameter. The `type` field points to a type structure indicating the type of the variable. The `name` field gives the name (obviously), and the `which` field gives the ordinal position of local variables and parameters. (More on that later.)

As with all the other data structures we have created so far, we must have a factory function like this:

```
struct symbol * symbol_create( symbol_t kind,
                               struct type *type,
                               char *name ) {
    struct symbol *s = malloc(sizeof(*s));
    s->kind = kind;
    s->type = type;
    s->name = name;
    return s;
}
```

To begin semantic analysis, we must create a suitable `symbol` structure for each variable declaration, and enter it into the symbol table.

Conceptually, the symbol table is just a map between the name of each variable, and the symbol structure that describes it:



However, it's not *quite* that simple, because most programming languages allow the same variable name to be used multiple times, as long as each definition is in a distinct **scope**. In C-like languages (including C-Minor) there is a global scope, a scope for function parameters and local variables, and then nested scopes everywhere curly braces appear.

For example, the following C-Minor program defines the symbol `x` three times, each with a different type and storage class. When run, the program should print `10 hello false`.

```
x: integer = 10;

f: function void ( x: string ) =
{
    print x, "\n";
    {
        x: boolean = false;
        print x, "\n";
    }
}

main: function void () =
{
    print x, "\n";
    f("hello");
}
```

To accomodate these multiple definitions, we will structure our symbol table as a stack of hash tables, as shown in Figure 7.2. Each hash table maps the names in a given scope to their corresponding symbols. This allows a symbol (like `x`) to exist in multiple scopes without conflict. As we process through the program, we will push a new table every time a scope is entered, and pop a table every time a scope is left.

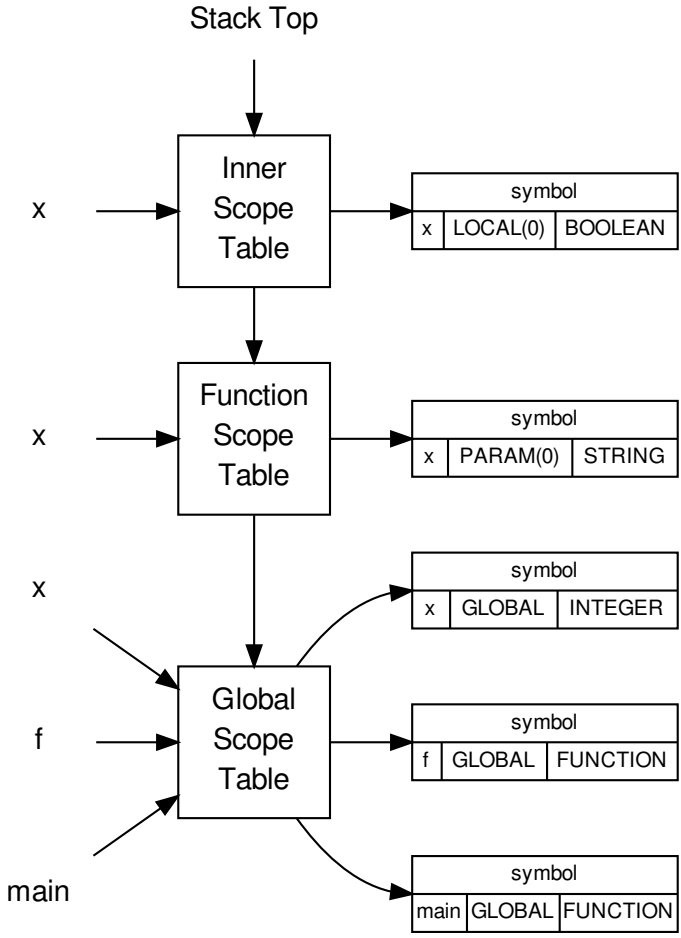


Figure 7.2: A Nested Symbol Table

```
void scope_enter();
void scope_exit();
int  scope_level();

void scope_bind( const char *name, struct symbol *sym );
struct symbol * scope_lookup( const char *name );
struct symbol * scope_lookup_current( const char *name );
```

Figure 7.3: Symbol Table API

To manipulate the symbol table, we define five operations in the API given in Figure 7.3. They have the following meaning:

- `scope_enter()` causes a new hash table to be pushed on the top of the stack, representing a new scope.
- `scope_leave()` causes the topmost hash table to be removed.
- `scope_level()` returns the number of hash tables in the current stack. (This is helpful to know whether we are at the global scope or not.)
- `scope_bind(name,sym)` adds an entry to the topmost hash table of the stack, mapping `name` to the symbol structure `sym`.
- `scope_lookup(name)` searches the stack of hash tables from top to bottom, looking for the first entry that matches `name` exactly. if no match is found, it returns null.
- `scope_lookup_current(name)` works like `scope_lookup` except that it only searches the topmost table. This is used to determine whether a symbol has already been defined in the current scope.

Exercise 6. Implement the symbol and scope functions in `symbol.c` and `scope.c`, using an existing hash table implementation as a starting point.

7.3 Name Resolution

With the symbol table in place, we are now ready to match each use of a variable name to its matching definition. This process is known as **name resolution**. To implement name resolution, we will write a `resolve` method for each of the structures in the AST, including `decl_resolve()`, `stmt_resolve()` and so forth.

Collectively, these methods must iterate over the entire AST, looking for variable declarations and uses. Wherever a variable is declared, it must be entered into the symbol table and the `symbol` structure linked into the AST. Wherever a variable is used, it must be looked up in the symbol table, and the `symbol` structure linked into the AST. Of course, if a symbol is declared twice in same scope, or used without declaration, then an appropriate error message must be emitted.

We will begin with declarations, as shown in Figure 7.4. Each `struct decl` represents a variable declaration of some kind, so `decl_resolve` will create a new symbol, and then bind it to the name of the declaration in the current scope. If the declaration represents an expression (`d->value` is not null) then the expression should be resolved. If the declaration represents a function (`d->code` is not null) then we must create a new scope and resolve the parameters and the code. Figure 7.4 gives some sample code for resolving declarations.

In a similar fashion, we must write resolve methods for each structure in the AST. `stmt_resolve()` (not shown) must simply call the appropriate `resolve` on each of its sub-components. In the case of a STMT_BLOCK, it must also enter and leave a new scope. `param_list_resolve()` (also not shown) must enter a new variable declaration for each parameter of a function, so that those definitions are available to the code of a function.

To perform name resolution on the entire AST, you may simply invoke `decl_resolve()` on the root node of the AST.

Exercise 7. Complete the name resolution code by writing `stmt_resolve()` and `param_list_resolve()` and any other supporting code needed.

Exercise 8. Modify `decl_resolve()` and `expr_resolve()` display errors when the same name is declared twice, or when a variables is used without a declaration.

```
void decl_resolve( struct decl *d )
{
    if(!d) return;

    symbol_t kind = scope_level() > 1 ?
                    SYMBOL_LOCAL : SYMBOL_GLOBAL;

    d->symbol = symbol_create(kind,d->type,d->name);

    scope_bind(d->name,d->symbol);

    if(d->value) {
        expr_resolve(d->value);
    }

    if(d->code) {
        scope_enter();
        param_list_resolve(d->type->params);
        stmt_resolve(d->code);
        scope_exit();
    }

    decl_resolve(d->next);
}
```

Figure 7.4: Name Resolution for Declarations

```
void expr_resolve( struct expr *e )
{
    if(!e) return;

    if( e->kind==EXPR_NAME ) {
        e->symbol = scope_lookup(e->name);
    } else {
       expr_resolve( e->left );
       expr_resolve( e->right );
    }
}
```

Figure 7.5: Name Resolution for Expressions