Introduction to Compilers and Language Design

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at this website:
`http://compilerbook.org`

Draft version: December 12, 2017

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!

Before we can perform type checking, we must determine the type of each identifier used in an expression. 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. We solve this problem by performing **name resolution**, in which each definition of a variable is entered into a **symbol table**. This table is referenced throughout the semantic analysis stage whenever we need to evaluate the correctness of some code.

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.

Semantic analysis also includes other forms of checking the correctness of a program, such as examining the limits of arrays, avoiding bad pointer traversals, and examining control flow. Depending on the design of the language, some of these problems can be detected at compile time, while others may need to wait until runtime.

7.1 Overview of Type Systems

Most programming languages assign to every value (whether a literal, constant, or variable) 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 most languages, these atomic 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:

- **Correctness.** 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 is 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[]`. As a result, the program could have almost any outcome, including incorrect output, silent data corruption, or an infinite loop.

```
/* 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 the basic structures of the language. 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 often used 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 X86 machines, `(a+b)` would be translated to an `ADDL` instruction for integers, but an `FMUL` instruction for floating point values. To know which instruction to apply, we must first determine the type of `a` and `b` and deduce the intended meaning of `+`.

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 instanceof Chair) {
       System.out.println("Sit up straight!\n");
   } else if ( f instanceof 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 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;
```

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

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.

7.2 Designing a Type System

To describe the type system of a language, we must explain its atomic types, its compound types, and the rules for assigning and converting between types.

The **atomic types** of a language are the simple types used to describe individual variables that are typically (though not always) stored in single registers in assembly language: integers, floating point numbers, boolean values, and so forth. For each atomic type, it is necessary to clearly define the range that is supported. For example, integers may be signed or unsigned, be 8 or 16 or 32 or 64 bits; floating point numbers could be 32 or 40 or 64 bits; characters could be ASCII or Unicode.

Many languages allow for **user-defined types** in which the programmer defines a new type that is implemented using an atomic type, but gives it a new meaning by restricting the range. For example, in Ada, you might define new types for days and months:

```
-- This is Ada code
type Day is range 1..31;
type Month is range 1..12;
```

This is useful because variables and functions dealing with days and months are now kept separate, preventing you from accidentally assigning one to another, or for giving the value 13 to a variable of type `Month`.

C has a similar feature, but it is much weaker: `typedef` declares a new name for a type, but doesn't have any means of restricting the range, and doesn't prevent you from making assignments between types that share the same base type:

```
/* This is C code */
typedef int Month;
typedef int Day;

/* Assigning m to d is allowed in C,
   because they are both integers. */

Month m = 10;
Day d = m;
```

Enumerations are another kind of user-defined type in which the programmer indicates a finite set of symbolic values that a variable can contain. For example, if you are working with uncertain boolean variables in Rust, you might declare:

```
/* This is Rust code */
enum Fuzzy { True, False, Uncertain };
```

Internally, an enumeration value is simply an integer, but it makes the source code more readable, and also allows the compiler to prevent the programmer from assigning an illegal value. Once again, the C language allows you to declare enumerations, but doesn't prevent you from mixing and matching integers and enumerations.

The **compound types** of a language combine together existing types into more complex aggregations. You are surely familiar with a **structure type** (or **record type**) that groups together several values into a larger whole. For example, you might group together latitude and longitude to treat them as a single `coordinate`:

```
/* This is Go code */
type coordinates struct {
    latitude float64
    longitude float64
}
```

Less frequently used are **union types** in which multiple symbols occupy the same memory. For example, in C, you can declare a union type of `number` that contains an overlapping float and integer:

```
/* This is C code */
union number {
int i;
float f;
};

union number n;
n.i = 10;
n.f = 3.14;
```

In this case, `n.i` and `n.f` occupy the same memory. If you assign `10` to `n.i` and read it back, you will see `10` as expected. However, if you assign `10` to `n.i` and read back `n.f`, you will likely observe a garbage value, depending on how exactly the two values are mapped into memory. Union types are occasionally handy when implementing operating system features such as device drivers, because hardware interfaces often re-use the same memory locations for multiple purposes.

Some languages provide a **variant type** which allows the programmer to explicitly describe a type with multiple variants, each with different fields. This is similar to the concept of a union type, but prevents the programmer from performing unsafe accesses. For example, Rust allows us to create a variant type representing an expression tree:

```
/* This is Rust code */
enum Expression {
    ADD{ left: Expression, right: Expression },
    MULTIPLY{ left: Expression, right: Expression },
    INTEGER{ value: i32 },
    NAME{ name: string }
}
```

This variant type is tightly controlled so that it is difficult to use incorrectly. For an Expression of type `ADD`, it has `left` and `right` fields which can be used in the expected way. For an Expression of type `NAME`, the `name` field can be used. The other fields are simply not available unless the appropriate type is selected.

Finally, we must define what happens when unlike types are used together. Suppose that an integer `i` is assigned to a floating point `f`. A similar question arises when an integer is passed to a function expecting a floating point as an argument. There are several possibilities for what a language may do in this case:

- **Disallow the assignment.** A very strict language (like C-Minor) could simply emit an error and prevent the program from compiling! Perhaps it simply makes no sense to make the assignment, and the compiler is saving the programmer from a grievous error. If the assignment is *really* desired, it could be accomplished by requiring that the programmer call a built-in conversion function (e.g. `IntToFloat`) that accepts one type and returns another.

- **Perform a bitwise copy.** If the two variables have the same underlying storage size, the unlike assignment could be accomplished by just copying the bits in one variable to the location of the other. This is *usually* a bad idea, since there is no guarantee that one data type has any meaning in the other context. But it does happen in a few select cases, such as when assigning different pointer types in C.

- **Convert to an equivalent value.** For certain types, the compiler may have built-in conversions that change the value to the desired type implicitly. For example, it is common to implicitly convert between integers and floating points, or between signed and unsigned integers. But this does not mean the operation is safe! An implied conversion can lose information, resulting in very tricky bugs.

- **Interpret the value in a different way.** In some cases, it may be desirable to convert the value into some that is not equivalent, but still useful for the programmer. For example, in Perl, when a list is copied to a scalar context, the *length* of the list is placed in the target variable, rather than the content of the list.

```
@days = ("Monday", "Tuesday", "Wednesday", ... );
@a = @days; # copies the array to array a
$b = @days; # puts the length of the array into b
```

7.3 The C-Minor Type System

The C-Minor type system is safe, static, and explicit. As a result, it is fairly compact to describe, and straightforward to implement, and will eliminate a large number of programming errors. However, it may be more strict than some languages, so there are going to be a large number of errors that we must detect.

C-Minor has the following atomic types:

- `integer` - A 64 bit signed integer.
- `boolean` - Limited to symbols `true` or `false`.
- `char` - Limited to ASCII values.
- `string` - ASCII values, null terminated.
- `void` - Only used for a function that returns no value.

And the following compound types:

- `array [size] type`
- `function type ( a: type, b: type, ... )`

And here are the type rules that must be enforced:

- A value may only be assigned to a variable of the same type.
- A function parameter may only accept a value of the same type.
- The type of a `return` statement must match the function return type.
- All binary operators must have the same type on the left and right hand sides.
- The equality operators `!=` and `==` may be applied to any type except `void`, `array`, or `function` and always return `boolean`.

- The comparison operators `< <= >= >` may only be applied to `integer` values and always return `boolean`.
- The boolean operators `! && ||` may only be applied to `boolean` values and always return `boolean`.
- The arithmetic operators `+ - * / % ^ ++ --` may only be applied to `integer` values and always return `integer`.

7.4 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

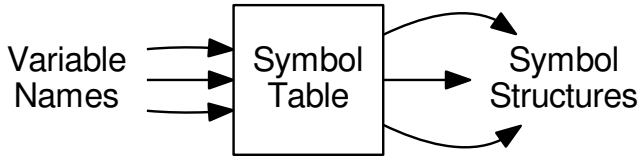
The `kind` field indicates whether the symbol is a local variable, a global variable, or a function parameter. The `type` field points to a type structure indicating the type of the variable. The `name` field gives the name (obviously), and the `which` field gives the ordinal position of local variables 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 accommodate 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 proceed 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.

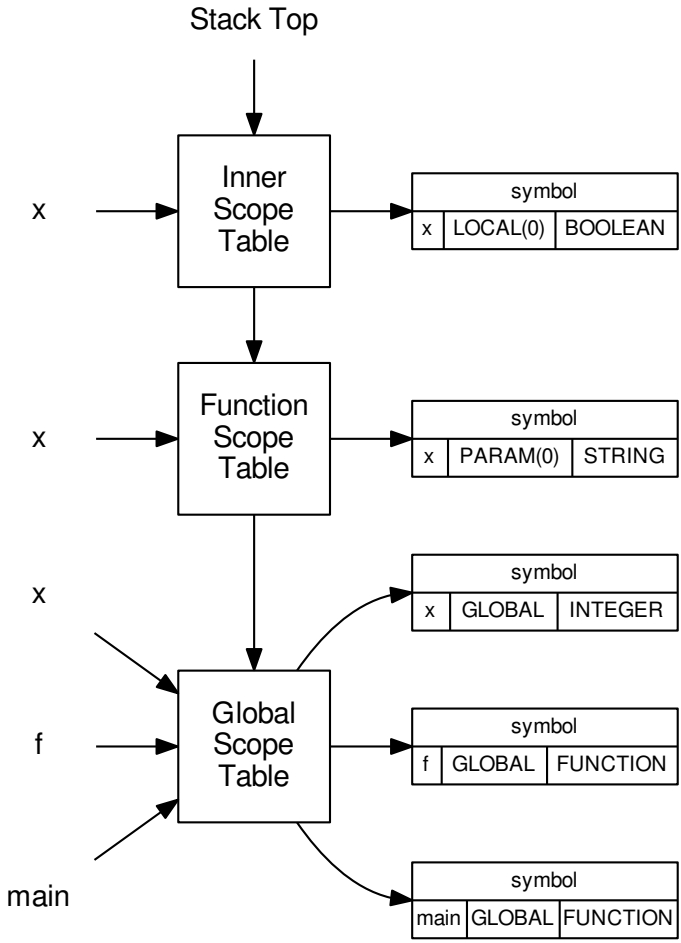


Figure 7.2: A Nested Symbol Table

```
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 six 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.

7.5 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. As always in this book, consider this starter code in order to give you the basic idea. You will have to make some changes in order to accommodate all the features of the language, handle errors cleanly, and so forth.

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()` once on the root node of the AST. This function will traverse the entire tree by calling the necessary sub-functions.

```
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);

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

    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

7.6 Implementing Type Checking

Before checking expressions, we need some helper functions for checking and manipulating type structures. Here is pseudo-code for checking equality, copying, and deleting types:

```
int type_equals( struct type *a, struct type *b )
{
    if( a->kind == b->kind ) {
        if( a and b are atomic types ){
            Return true;
        } else if ( both are array ) {
            Return true if subtype is recursively equal
        } else if ( both are function ) {
            Return true if both subtype and params
            are recursively equal
        }
    } else {
        return false;
    }
}

struct type * type_copy( struct type *t )
{
    Return a duplicate copy of t, making sure
    to duplicate subtype and params recursively.
}

void type_delete( struct type *t )
{
    Free all the elements of t recursively.
}
```

Next, we construct a function `expr_typecheck` that will compute the proper type of an expression, and return it. To simplify our code, we assert that `expr_typecheck`, if called on a non-null `expr`, will always return a newly-allocated `type` structure. If the expression contains an invalid combination of types, then `expr_typecheck` will print out an error, but return a valid type, so that the compiler can continue on and find as many errors as possible.

The general approach is to perform a recursive, post-order traversal of the expression tree. At the leaves of the tree, the type of the node simply corresponds to the kind of the expression node: an integer literal has integer type, a string literal has string type, and so on. If we encounter a variable name, the type can be determined by following the `symbol` pointer

to the symbol structure, which contains the type. This type is copied and returned to the parent node.

For interior nodes of the expression tree, we must compare the type of the left and right subtrees, and determine if they are compatible with the rules indicated in Section 7.3. If not, we emit an error message and increment a global error counter. Either way, we return the appropriate type for the operator. The types of the left and right branches are no longer needed and can be deleted before returning.

Here is the basic code structure:

```
struct type * expr_typecheck( struct expr *e )
{
    if(!e) return;

    struct type *lt = expr_typecheck(e->left);
    struct type *rt = expr_typecheck(e->right);

    struct type *result;

    switch(e->kind) {
        case EXPR_INTEGER_LITERAL:
            result = type_create(TYPE_INTEGER,0,0);
            break;
        case EXPR_STRING_LITERAL:
            result = type_create(TYPE_STRING,0,0);
            break;


        /* more cases here */


    }

    type_delete(lt);
    type_delete(rt);

    return result;
}
```

Let's consider the cases for a few operators in detail. Arithmetic operators can only be applied to integers, and always return an integer type:

```
case EXPR_ADD:
    if( lt->kind != TYPE_INTEGER || rt->kind!=TYPE_INTEGER ) {
        /* display an error */
    }
    result = type_create(TYPE_INTEGER,0,0);
    break;
```

The equality operators can be applied to most types, as long as the types are equal on both sides. These always return boolean.

```
case EXPR_EQ:
case EXPR_NE:
    if(!type_equals(lt,rt)) {
        /* display an error */
    }
    if(lt->kind==TYPE_VOID ||
       lt->kind==TYPE_ARRAY ||
       lt->kind==TYPE_FUNCTION) {
        /* display an error */
    }
    result = type_create(TYPE_BOOLEAN,0,0);
    break;
```

An array derefence like `a[i]` requires that `a` be an array, `i` be an integer, and returns the subtype of the array:

```
case EXPR_DEREF:
    if(lt->kind==TYPE_ARRAY) {
        if(rt->kind!=TYPE_INTEGER) {
            /* error: index not an integer */
        }
        result = type_copy(lt->subtype);
    } else {
        /* error: not an array */
        /* but we need to return a valid type */
        result = type_copy(lt);
    }
    break;
```

Most of the hard work in typechecking is done in `expr_typecheck`, but we still need to implement typechecking on declarations, statements, and the other elements of the AST. `decl_typecheck`, `stmt_typecheck` and the other typechecking methods simply traverse the AST, compute the

type of expressions, and then check them against declarations and other constraints as needed.

For example, `decl_typecheck` simply confirms that variable declarations match their initializers and otherwise typechecks the body of function declarations:

```
void decl_typecheck( struct decl *d )
{
    if( d->value ) {
struct type *t;
        t = expr_typecheck(d->value);
        if(!type_equals(t,d->symbol->type)) {
            /* display an error */
        }
    }
    if(d->code) {
        stmt_typecheck(d->code);
    }
}
```

Statements must be typechecked by evaluating each of their components, and then verifying that types match where needed. After the type is examined, it is no longer needed and may be deleted. For example, if-then statements require that the control expression have boolean type:

```
void stmt_typecheck( struct stmt *s )
{
    struct type *t;
    switch(s->kind) {
        case STMT_EXPR:
            t = expr_typecheck(s->expr);
            type_delete(t);
            break;
        case STMT_IF_THEN:
            t = expr_typecheck(s->expr);
            if(t->kind!=TYPE_BOOLEAN) {
                /* display an error */
            }
            type_delete(t);
            stmt_typecheck(s->body);
            stmt_typecheck(s->else_body);
            break;

        /* more cases here */
    }
}
```

7.7 Error Messages

Compilers in general are notorious for displaying terrible error messages. Fortunately, we have developed enough code structure that it is straightforward to display an informative error message that explains exactly what types were discovered, and what the problem is.

For example, this bit of C-Minor code has a mess of type problems:

```
s: string = "hello";
b: boolean = false;
i: integer = s + (b<5);
```

Most compilers would emit an unhelpful message like this:

```
error: type compatibility in expression
```

But, your project compiler can very easily have much more detailed error messages like this:

```
error: cannot compare a boolean (b) to an integer (5)
error: cannot add a boolean (b<5) to a string (s)
```

It's just a matter of taking some care in printing out each of the expressions and types involved when a problem is found:

```
printf("error: cannot add a ");
type_print(lt);
printf(" (");
expr_print(e->left);
printf(") to a ");
type_print(rt);
printf(" (");
expr_print(e->right);
printf(")\n");
```

7.8 Exercises

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

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

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

4. Complete the implementation of `expr_typecheck` so that it checks and returns the type of all kinds of expressions.

5. Complete the implementation of `stmt_typecheck` by enforcing the constraints particularly to each kind of statement.

6. Write a function `myprintf` that displays printf-style format strings, but supports symbols like `%T` for types, `%E` for expressions, and so forth. This will make it easier to emit error messages, like this:

   ```
   myprintf("error: cannot add a %T (%E) to a %T (%E)\n",
            lt,e->left,rt,e->right);
   ```

   Consult a standard C manual and learn about the functions in the `stdarg.h` header for creating variadic functions.