Disclaimer
This handout is not perfect; corrections may be made. Updates and major clarifications will be incorporated in this document and noted on Piazza as they are made. Check for updates regularly.

So, what isn’t going to be tested?
Wait! Before asking a question about WNBT stuff on Piazza, see the “What we’re not testing or using in test cases” section a few sections below to see if it has already been answered.

Error messages, which order? How many?
If a check has a list of multiple bullet points to check, check them in the order of the bullet points. Report only the first error occurring in the list.

Important notes about turn-in
Refer to the turn-in procedure document on the website for instructions on the turn-in procedure.

IMPORTANT: Remove all debugging output before you turn-in your project. Failure to do so will result in a very poor score.

IMPORTANT: Do a test turn in about a week before the deadline. This way we can help resolve turn-in problems early and ensure a smooth turn-in process when the deadline approaches. The two commands that need to work without any errors for a proper turn-in are:

```
made new
./RC someTestFile.rc
```

Background
In this assignment you will implement the semantic checker component of your compiler.
Semantic checking involves storing information in a symbol table (usually during declarations) and then accessing that information (usually in statements) in order to verify that the input conforms to our semantic rules.

For the purposes of this assignment, we have defined a new language called RC (reduced-C). The language is similar to a somewhat watered-down version of C with some twists here and there. The project is divided into 4 phases. The phases give you an indication of (roughly) how long it should take for each part to be completed, as well as show the grading breakdown. You should complete the phases in chronological order.

Error messages

As in all compilers, the error messages must be precisely specified. Error messages are provided in ErrorMsg.java. Do not modify the messages and do not create your own.

The error messages in ErrorMsg.java have the place holders %O, %T, %D, and %S for operators, types, integer values, and identifiers/object names, respectively. These place holders indicate what to pass in to Formatter.toString() for a particular error message. For example, you could pass in “+” for %O, “bool” for %T, “5” for %D, and “myIntVar” for %S.

All error messages should be printed to standard output, not standard error.

Remember to increment the error count whenever you print out an error. The starter code already does this so refer to it for examples. All error messages will be preceded by a line of text specifying the current filename (this is already done by the starter code):
Error, "file.rc":
   error message

Note that the filename is on a separate (preceding) line from the error message.

Running the project

The basic steps to compile and run the project from the command line are:

```
make
./RC someTestFile.rc
```

All the Java source files and the rc.cup source file are found under src/. The generated Java class files are put under bin/. The RC and RCdbg files are simple scripts that run your Java project.

To enable error message line numbers for debugging purposes, do

```
make debug
```
If you want to use Eclipse to work on the project, see the instructions in build.xml included in the starter code. You can either complete the project using the command line and the provided Makefile or using Eclipse with the build.xml ant buildfile. However, if you choose to work with Eclipse, you should make sure you can still compile and run with the Makefile to ensure that the turn in process works correctly (as mentioned above.)

Multiple and cascading errors

While a single statement may contain many errors, it is difficult to specify exactly how these errors are reported. To simplify your task:

1. You should report only the first error encountered in the parse in any simple statement (e.g. assignment) or any part of a multi-part statement (e.g. the test expression of a while statement), ignoring any further errors until the end of that statement or statement part.

   If a check has a list of multiple bullet points to check, check them in the order of the bullet points. Report only the first error occurring in the list.

Another way to tell which error to print first is to check the order of error messages listed in ErrorMsg.java. For example, the expression `++true` has two errors (operand is not a numeric type, and operand is not a modifiable lval), but only the first one (not a numeric type) should be reported.

Note: For some simple statements, it is not easily possible to avoid printing multiple errors. One such expression is `*a + *a`, where the type of variable `a` is not a pointer type. Such expressions will not be tested.

2. No variable, function, or type whose declaration contained an error will be used in the remainder of any test case. No identifier, having been declared erroneously once, will be re-declared.

Printing types in errors

Many error messages include a type name (`%T`). The printed forms of arrays, pointers, structs, and typedefs should obey the following guidelines.

1. Printing basic types

   Print integer types as "int", boolean types as "bool", and floating point types as "float".

2. Printing array types
Printed array types include the dimension size in brackets without any spaces (e.g. "int[10]" or "bool[2]").

3. Printing pointer types

Printed pointer types include the asterisk(s) without any spaces (e.g. "int*" or "float***"). Furthermore, when combined with arrays, the array dimension occurs after the asterisks (e.g. "int*[4]" for a variable that is an array of 4 integer pointers). The type of the nullptr keyword should be printed as "nullptr".

4. Printing typedefs

Variables defined using a typedef type should be printed using the name of the typedef, e.g.

```c
typedef int MYINTTYPE;
MYINTTYPE x;
```

The type of “x” should print "MYINTTYPE", not "int".

5. Printing structs

Struct definitions are only done via a structdef (a glorified typedef for structs). The only way a variable can be of a struct type is via the usage of the struct’s identifier in the declaration. Thus, structs follow the same rule as typedefs above, where the name (the identifier) of the struct definition is printed. So for this example

```c
structdef MYSTRUCT {
    float field1, field2;
};
MYSTRUCT y;
```

the type of “y” should print "MYSTRUCT".

6. Printing void types

When printing the type of a function call’s return value, if the function was declared as void, you should print type "void".

7. Printing error types

If an error type somehow surfaces in an error message which requires its type to be printed, print "ERROR".

What we’re not testing or using in test cases

- Syntax errors WNBT, i.e. all code fed to your compiler will be syntactically correct. However, semantic errors will be tested. (That’s the whole point of project 1.)
- The for loop and foreach loop constructs. But, you ARE responsible for the while loop.
- Redeclared or undeclared id errors (the starter code already handles these).
- Nested function declarations or structdef inside functions.
- The cin and cout built-in functions (but we will use these in the next project).
The `extern` keyword for functions/variables (but we will use this in the next project).
Constant function pointers.
Comparisons of function pointers to one another.
Assignment expressions where the left hand side is an expression of array type. But, you ARE responsible if the left hand side is an array identifier. (see below)

```c
int [6] arr;
int [6] arr2;
arr2 = arr1;  //this could be tested and it should generate an error
typedef int [6] MYARR;
typedef MYARR * PTRARR;
MYARR2 ptr;
*ptr = arr2 //won’t be tested. *ptr is an expression of array type
```

Multidimensional arrays (we ARE testing arrays of arrays, shown below)

```c
typedef int [6] MYARR;
typedef MYARR [4] MYARR2;
MYARR2 myArrayOfArray;

function : int main() {
    myArrayOfArray[3][5] = 4;  // tested – assignment to last element
    return 0;
}
```

Returning arrays by reference WNBT.
Passing expressions containing errors as an argument to a function call.

```c
void foo(int x) { ... }

foo(2+true) // WNBT
foo(true)   // Will be tested, type mismatch
```

Constant arrays and pointers WNBT.
Comparing function pointers WNBT.
Address-of on array names WNBT.
The assignment

Your task for this assignment is to implement the following semantic checks. Note that frequently the terms “equivalent” and “assignable” are used – these will be discussed in more detail in lecture and in discussion section.

For convenience, we generally use the term equivalent to mean equal types.

The term assignable includes the class of equivalent types, as well as any implicit type coercions allowed. For this project, the only implicit type coercions are promoting an integer to a float and coercing an array type into a pointer type.

The following table shows some examples:

<table>
<thead>
<tr>
<th>Term</th>
<th>Types</th>
</tr>
</thead>
</table>
| Equivalent | int <-> int  
|          | float <-> float  
|          | bool <-> bool  
|          | int** <-> int**  
|          | Type[5] <-> Type[5] |
| Assignable | Everything in equivalent  
|           | int -> float (only in one direction)  
|           | TYPE[5] -> TYPE* (only in one direction) |

The rule TYPE[5] -> TYPE* means that for any given base type TYPE, an array of TYPE is assignable to a pointer to TYPE. In other words, base types need to be equivalent.

For example:

```plaintext
int[6] iarr;
float[6] farr;
float * ptr;
ptr = farr;  // ok
ptr = iarr;  // should generate error
```

Note: type void is not equivalent to anything (even itself). Using an object of type void in any expression should result in an error.

Additionally, the terms “modifiable L-value”, “non-modifiable L-value”, and “R-value” are used frequently. L-values are object locators. The difference between a modifiable L-value and a non-modifiable L-value is that the latter cannot be modified.
The statement “is not a modifiable L-value” is *not* the same as saying “is a non-modifiable L-value”. The difference is this:

- **“Not a modifiable L-value”** means something is either not addressable, or not modifiable, or neither addressable nor modifiable.
- **“Non-modifiable L-value”** means it is addressable, but not modifiable. Another point of confusion is that something that is “not modifiable” is *not* necessarily a constant value. The following table shows the definitions and examples:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifiable L-value</td>
<td>Addressable and modifiable</td>
<td>● Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● The results of expressions with the operands * . . -&gt; [ ] (Note: array names and function names are never modifiable lvals, even if access using . or -&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Function call expressions where the function returns by reference</td>
</tr>
<tr>
<td>Non-modifiable L-value</td>
<td>Addressable, but not modifiable</td>
<td>● Declared constants (e.g. const int x = 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● The name of an array</td>
</tr>
<tr>
<td>R-value</td>
<td>Neither addressable nor modifiable</td>
<td>● The results from arithmetic operations (e.g. x+y)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Constant literals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● The name of a function (which is a function pointer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Results of address-of and type casts</td>
</tr>
</tbody>
</table>

For the purposes of this assignment, the following rules apply:

- All types (except structs) use **structural equivalence**.
- All structdefs use **strict name equivalence**.
- All typedefs use **loose name equivalence** to resolve down to the lowest-level. For example, in the example below, the types INTEGER and MONTH both resolve down to int, so when comparing objects of type INTEGER and MONTH, you would use the int type.
- Struct-level operations (e.g. assignment, equality, and inequality) use **name equivalence**. All structs are defined with the structdef keyword.
- Array identifiers are **non-modifiable L-values** (they are a constant pointer to the first element in the array).
- Struct identifiers are **modifiable L-values**.

Here is an example illustrating some of these points:

typedef int INTEGER;
typedef int MONTH;
INTEGER i;
MONTH m;
float r;

structdef REC1 { float a; }
structdef REC2 { float a; }
typedef REC1 REC3;

REC1 r1;
REC2 r2;
REC3 r3;

function : int f(REC1 &a) { /* stuff */ }

float[5] a1;
int[5] a2;

function : int g(float[5] &a) { /* stuff */ }

int* p1;
INTEGER* p2;
REC1* p3;
REC2* p4;
REC3* p5;

function : int main() {

i = m;  // okay, assignable – loose name equivalent
i = r;  // error, not assignable – float cannot be assigned to int
    // Error Message: error3b_Assign

r = i;  // okay, assignable – int can be assigned to float (coercion)

f(r1);  // okay, same type/strict name equivalent
f(r2);  // error, not strict name equivalent
    // Error Message: error5r_Call

f(r3);  // okay, same type/loose name equivalent typedef then
    // strict name equivalent on the struct types

g(a1);  // okay, structurally equivalent
g(a2);  // error, not structurally equivalent
    // Error Message: error5r_Call

a1 = a1;  // error, arrays are not modifiable L-vals
    // Error Message: error3a_Assign

r1 = r1;  // okay, strict name equiv & struct variables are mod L-vals
r1 = r2;  // error, not strict name equivalent
    // Error Message: error3b_Assign
r3 = r1; // okay, loose name equivalent typedef then strict name
// equiv on the struct types & struct variables are mod L-vals
p1 = p2; // okay, types pointed to are structurally equivalent
p3 = p4; // error, types pointed to (structs) are not strict name equiv
// Error Message: error3b_Assign
p3 = p5; // okay, types pointed to (structs) are strict name equivalent
return 0;
}

When a check fails, your compiler is expected to keep checking subsequent expressions or statements. Your compiler should not crash or terminate on any of the semantic checks in this project.

Note that the time periods for the various phases are estimates to help you plan your time -- they are not due dates. The percentages are also approximate and are provided to help gauge the impact of each phase.

Phase 0 (1st week)
1. Edit the grammar file (rc.cup) to support the following new rules.
   1. new x;
   2. delete x;

Here are the rules you will need to add to your grammar:

```
NewStmt -> T_NEW Designator T_SEMI
DeleteStmt -> T_DELETE Designator T_SEMI
```

You also need to add other associated terminal/non-terminal symbols in the grammar file and lexer file (Lexer.java). Once this is completed, an example program (named new.rc in the starter code directory) should run without syntax errors.

2. There are two bugs in the starter code.
   1. The method access() in SymbolTable.java performs a bottom-up FIFO search of its scope stack (class Stack is a Vector [extends Vector] -- terrible design paradigm -- should use composition and not inheritance, but that is another topic). The scope stack should be searched from the top-down as a LIFO.
      A search for an identifier with both global and local scope (global variable x and local variable x) will incorrectly find the global scoped entry first instead of correctly finding the local scoped entry first.
      Fix SymbolTable.access() so it does the right thing. An example program
(named scope.rc) is provided in the starter code to illustrate the problem. You will be able to tell if your bug fix is correct after completing Check #3 if scope.rc results in only one error being generated for the expression \( y = 2 \).

2. RC does not currently allow unary plus and minus expressions (e.g. \(-x\) as in \(2 \times -x\)), but the grammar includes a rule, UnarySign, to allow this. Enable this expression by defining the UnarySign rule.

   We will use UnarySign with simple numeric types/constants/expressions only (e.g. \(-\text{var} \) or \(-5\) or \(+17\) or \(-(10+2)\)) and not with non-numeric types (-false or +nullptr) so you do not need to check for illegal uses of UnarySign.

3. Hex and Octal
   
   Integer literals that are hexadecimal (0x42 or 0X42), and octal (042). This will be tested extensively.
Phase I (40% -- 1 week)

Declarations, statements, and expressions consisting of variables and literals of a basic type (int/float/bool), functions (global), and exits.

Also, just like in C, in variable and constant declarations the pointer and array tokens bind to the id, not the main type. This distinction is important for multivariable declarations.

For example:

```c
int * x, y, [42] z;
```

Here, \( x \) is of type \( int * \), \( y \) is of type \( int \), and \( z \) is of type \( int[42] \);

You will be responsible for implementing the auto keyword, a C++11 feature that allows the compiler to infer the type of a variable from the type of the initializer. Instead of declaring a variable with a specific type you can declare it with auto. Variables declared with auto can also be declared as static and const.

```c
auto i = 5; // i is of type int
static auto b = true; // b is of type bool
const auto f = 1.0; // f is of type float
```

Note that once an auto variable’s type has been inferred, that inferred type is the type of the variable. You should use the inferred type in printing future error messages, etc, and not “auto”.

Check #0:

Detect **undeclared global identifiers** accessed with the scope resolution operator :: For example, the following code should generate an error because identifier \( x \) is not found in the global scope:

```c
int myX;
function : void foo(){
    int x;
    int i = 5 + ::x;  // Error, cannot find x in the global scope
}
```

Check #1:

Detect a **type conflict in an expression** -- that is, expressions of the form

\[ x \ OP \ y \]

where the types of either \( x \) or \( y \) are incompatible with \( OP \). The valid types (and resultant types) are as follows:
For the T_PLUS, T_MINUS, T_STAR, T_SLASH operators, the operand types must be numeric (equivalent to either int or float), and the resulting type is int when both operands are int, or float otherwise.

For the T_MOD operator, the operand types must be equivalent to int, and the resulting type is int.

For the T_LT, T_LTE, T_GT, and T_GTE operators, the operand types must be numeric, and the resulting type is bool.

For the T_EQU and T_NEQ operators, the operand types must be either BOTH numeric, or BOTH equivalent to bool, and the resulting type is bool.

For the T_OR, T_AND, and T_NOT operators, the operand types must be equivalent to bool, and the resulting type is bool. Note: T_NOT is a unary operator.

For the T_AMPERSAND, T_CARET, and T_BAR bitwise operators, the operand types must be equivalent to int, and the resulting type is int.

Check the left operand first, and then the second. In Phase I, the operands are restricted to simple variables of basic type (int, float, bool) and literals, including the UnarySign (this will be extended in the later phases).

Note: The result of any arithmetic or logical operation is an R-value.

Check #2:
Detect a type conflict in a pre/post increment/decrement -- that is, for expressions like

\[
\begin{align*}
y &= x++; \\
w &= x++ + --y;
\end{align*}
\]

an error should be generated if

- the type of the operand to the increment or decrement is not of type int or float (numeric);
- the operand is not a modifiable L-value.

The resulting object should be marked as an R-value.

Note: in Phase I, only int and float types are considered valid. This will be extended in the later phases to allow pointer types as well (hence the error message including pointers).

Check #3a:
Detect an illegal assignment -- that is, an assignment of the form

\[
\text{myA} = \text{Expr}
\]

where myA is not a modifiable L-value. Some examples (most of which are implemented later in this project) include typedefs, structdefs, function names, function calls, constants (literals or
declared), array designators, results of the address-of operation, and results of type casts. Later
in the project, resulting expressions from *, ., ->, and [] will be incorporated as valid modifiable
L-values.

Note: The case where myA is a type name is already caught by the starter code as a syntax
error and does not need to be modified (please leave the message that is originally printed in
this case intact).

Note: Expr will only be either simple arithmetic expressions (including UnarySign) or another
assignment expression for this check. More complicated expressions (including pointer
dereferences) will be done in phases II and III. The expression resulting from an assignment
should be marked as an R-value.

For a chain of assignment expressions in a statement, only report the first error encountered
due to an illegal assignment, ignoring any subsequent illegal assignments in the same
expression. For example, the expression below should print only one error, resulting from
trying to assign 4 = 2. The subsequent illegal assignments to the left of the chain should not
produce an error. This is because the assignment operator is right-associative.

\[ 1 = 3 = 4 = 2; \]

Note: A partially functional error check already exists in MyParser.java in the DoAssignExpr()
method. You need to extend this check to display the proper message.

Check #3b:
Detect a type conflict in an assignment -- that is, an assignment of the form
\[ x = y \]
where the type of \( y \) is not assignable to the type of \( x \).

Another example:

```java
static float f = 5.5;
function : void foo()
{
  static int f;
  f = ::f; // error, not assignable: float cannot be assigned to int
}
```

Check #3c:
Implement decltype, that is, type expressions of the form:

```
decltype(Expr)
```
where the resulting type is the type of the $Expr$. $Expr$ is not run, so no side effects ever occur even if present in the expression. For example:

```c
int x = 10;
dcltype(x++) y; // y is a variable of type int
```

The value of $x$ is still 10.

Another example:

```c
int x = 10;
float y = 20;
function : dcltype(x+20.0) foo()
{
    return y;
}
```

$foo$ is a function with a return type of float.

You also need to add associated terminal/non-terminal symbols in the grammar file and lexer file.

**Check #4:**
Detect a type conflict in an if/while statement, that is, statements of the form

```c
if (Expr) { ... }
```

```c
while (Expr) { ... }
```

where the type of $Expr$ is not equivalent to bool or is not equivalent to int.

**Check #5:**
Detect an illegal function call. Errors should be generated if

- The number of arguments differs from the number of expected parameters;
- A parameter is declared as pass-by-value (default) and the corresponding argument's type is not assignable to the parameter type;
- A parameter is declared as pass-by-reference (using the &) and the corresponding argument's type is not equivalent to the parameter type;
- A parameter is declared as pass-by-reference and the corresponding argument is not a modifiable L-value.

If there is a problem with multiple arguments in a single function call, then an error should be generated for each such argument, in the order that the arguments are passed.
For now, functions can have any basic return type (int, float, bool), or void. In later checks, pointers and structs are valid return types, too. If the function call is used within an expression, the function’s return type should be used to do semantic checking within the expression. For example:

```plaintext
function : bool foo() { return false; }
function : void bar() { /* do nothing */ }
function : void main() {
    int x;
    x = x + foo();  // error: bool incompatible with + operator
    x = bar();      // error: void not assignable to int.
}
```

Note that a function’s type is not the same thing as its return type. A function’s type is a funcptr (see Check 18) that includes the function prototype information (return type and reference status and parameter types and reference status). Calls to functions that return by reference should be marked as modifiable L-values. Calls to functions that are NOT return-by-reference should be marked as R-values.

**Note:** Overloaded functions are extra credit.

**Check #6a:**
Detect an illegal return statement where no Expr is specified and the return type is not void.

```plaintext
return; // Where no Expr is specified and the return type is not void
```

**Check #6b:**
Detect an illegal return statement

For functions declared to return by value, an error should be generated if

- the type of return expression is not assignable to the return type of the function.

For functions declared to return by reference, an error should be generated if

- the type of the return expression is not equivalent to the return type of the function;
- the return expression is not a modifiable L-value.

**Note:** Return-by-reference will only be tested with non-void and non-array return types.

**Check #6c:**
Detect a missing return statement.
For functions declared with a non-void return type, at least one return statement (legal or illegal) must appear at the top level (i.e. not within an if or while statement).

Check #7:
Detect an illegal exit statement -- that is, statements of the form
\[ \text{exit(Expr);} \]
where \text{Expr} is not assignable to an int.
Phase II (40% -- 1 week)
Aliases, constant folding, arrays, break/continue, and structs, in addition to Phase I tasks.

Check #8a:
Detect an **illegal global/static runtime initialization**
```c
int x = Expr1;
static int y = Expr2;

function : void main() {
    static int z = Expr3;
}
```
For this example, an error should be generated if:
- The value of `Expr1` or `Expr2` or `Expr3` is not known at compile time.

**Note:** Local variables can be initialized at runtime and so this check does not apply to them.

Check #8b:
Detect an **illegal constant/variable initialization**
```c
Const Type c = ConstExpr;
Type x = Expr;
```
For this example error should be generated if:
- The value of `ConstExpr` is not known at compile time;
- The type of `ConstExpr` or `Expr` is not assignable to `Type`.

**Constant folding is required and will be checked.** When dealing with constant folding, if an arithmetic exception occurs (e.g. dividing by zero i.e. 0 or 0.0), the resulting constant value is irrelevant (since the object will now be an ErrorSTO) and the appropriate constant folding error message should be displayed. In this case, do not print the first error message listed above (constant initialization value not known at compile time). Note that you should throw an arithmetic exception for dividing by a zero float i.e. 0.0, even though in java, dividing by 0.0 might result in a NaN.

Check #9:
Extend all previous checks to allow for operands consisting of
- Named constants (example: `const int Zero = 0`)
- Variables of typedef types (example: `typedef int MYINT; MYINT x;`)
In other words, the rules for the previous checks are the same, but in Phase II we allow more complex expression operands. The following is a very simple example:

```c
typedef int INTEGER;

INTEGER a;
INTEGER b;

const int c = 2 + 3 * 0 - 1;

function : int main() {
    a = b + c;
    return a;
}
```

Check #10:
Detect an *illegal array declaration*.
Given a type declaration such as

```
Type[Index] List1;
```

An error should be generated if
- the type of the index expression (`Index` in this case) is not *equivalent* to `int`;
- the value of the index expression is not known at compile time (i.e., not a constant);
- the value of the index expression is not greater than 0.

**Note:** `Type` can be another array type or a typedef

```c
typedef float[10] ARR;

float[10] a;
ARR b;
ARR[2] c;      // array of array
```

Check #11:
Detect an *illegal array usage*.
Given a designator such as

```
MyList [ nIndex ]
or
MyList [ nIndex ][ nIndex ]
or
MyList [ nIndex ] ... [ nIndex ]   // Any number of [ nIndex ]s
```

an error should be generated if
The type of the designator preceding any [] operator is not an array or pointer type;

The type of the index expression (nIndex in this case) is not equivalent to int;

If the index expression is a constant, an error should be generated if the index is outside the bounds of the array (does not apply when the designator preceding the [] is of pointer type). We will be testing expressions like:
   a[55] or a[0-99] or a[x + 10] or a[c] or a[c+5] or a[-9]
   a[5][7] or a[-9][0] or a[x+4][c+2] or a[-c][5+3]

where c is a constant. If the index expression is a constant expression, you need to check that it is within the range 0 – (#-of-elements – 1) for that dimension. If it is not a constant expression (e.g., it is an integer variable), do not check its range (this will be done in Project II at run time).

Obviously, if the designator preceding the [] is a pointer type, no bounds checking will occur since the size is unknown.

**Note:** Extend all previous checks to include array designators in expressions (such as x[i] or y[i][j]).

```c
const int cc = 1;
typedef int[10] ARR;

int[10] a;
ARR b;
ARR[2] c; // This results in an array with 2 rows, 10 cols
int * d;

c[1][9] = a[2];
a[2] = d[0];
```

**Note:** Arrays will be passed-by-reference to array parameters of functions. Additionally, arrays will be passed-by-value to pointer parameters of functions. You will still need to check that the argument and parameter types are compatible. **Note that although array names are non-modifiable lvals, you should still allow them to be passed to reference parameters as arguments.**

**Check #12:** Detect an illegal break or continue statement

```c
break;
or
continue;
```

Errors should be generated if either statement is not within the body of a `while` loop.
Check #13a:
Detect an illegal struct declaration -- the same identifier twice in the same struct declaration.
If a field is duplicated multiple times, an error is reported for each duplicate instance:

```c
structdef MYS {
    int x, y;
    int x;   // duplicate id x, error #1
    int z, x; // duplicate id x, error #2
    float y; // duplicate id y, error #3
    int w;
    int w;   // duplicate id w, error #4
};
```

Check #13b:
Detect an illegal struct declaration -- invalid recursive struct definition

An error should be generated for invalid recursive struct definitions (i.e. containing recursive fields whose size cannot be determined at compile-time) like the one below:

```c
structdef MYSTRUCT {
    MYSTRUCT myRecursiveStruct;
};
```

**Note:** Recursive struct definitions using a pointer to the struct type are valid (should not produce an error) and will be tested:

```c
structdef MYSTRUCT {
    MYSTRUCT* myRecursivePtr;
};
```

Check #14:
Detect an illegal struct usage.

Given a designator such as

```
MyStruct.SomeField
```

an error should be generated if
- the type of `MyStruct` is not a struct type;
- the type of `MyStruct` has no field named `SomeField`.  

Phase III (20% -- 1 week)
Pointers, function pointers, sizeof, type casts, address-of

Check #15a:
Extend all previous checks to allow for operands consisting of
- Functions with pointer return types and struct return types.
- Structs and pointers, and pointer dereferences.

**Note:** we will be testing pointer dereferences such as:
```
y = (*ptr).x;
w = ptr->x;   // The arrow operator is basically the same as above
z = *ptr2;
```

An error should be generated if
- The type of `ptr` is not a pointer type for the `*` operator.
- The type of `ptr` is not a pointer to a struct for the `->` operator.

**Note:** For the arrow operator, first check if the left side is a pointer to a struct, using the error message for this check. Then check if the right side is a field within the struct, using the message from check #14 if necessary (when using this message for the arrow operator, the type argument is the *dereferenced* pointer’s type).

**Note:** Structs could be passed-by-reference or passed-by value to functions.

Check #15b:
Extend check #3 to allow pointer dereferences on the left-hand side of an assignment statement.
```
(*ptr).x = y;
ptr->x = w;
*ptr2 = z;
```

Check #15c:
Extend check #8 to detect an *illegal initialization of pointers*

For variable pointers (e.g. non constants), they can be initialized to nullptr, some other pointer, the result of an address-of operation (discussed in check #21), or the name of an array. Note: the array base type must be equivalent to the pointer base type. *Constant pointers will not be*
tested. Use the same error messages from check #8. Here are some examples:

    int x;
    int[4] a;
    int* r = &x;
    int* s = a;

Check #16:
Detect an *illegal new statement or illegal delete statement*—that is, for a statement of the form

    new x;

or

    delete x;

an error should be generated if

- x is not a modifiable L-value;
- x is not of a valid pointer type.

**Note:** In Phase 0 you made changes to your grammar file in order to support the call to new x
and delete x.

Check #17:
Extend Checks #1-3 to allow for operands consisting of objects of pointer type, for the following operators:

- For the T_EQU and T_NEQ operators (see Check #1), the operand types must BOTH be of equivalent pointer type or one is of pointer type and the other is of type nullptr.
  - If both operands are nullptr, a constant expression is returned. The resulting type is bool.
- For the pre/post increment/decrement (see Check #2), allow for operands of type equivalent to a pointer type too. The error message for Check #2 already handles pointers.
- Assignment compatibility of variables and values of pointer type (see Check #3).
  - nullptr should be treated as a constant assignable to variables of any pointer type.

**Note:** For the cases with two operands (T_EQU and T_NEQ), if either operand is a pointer type or nullptr, use error messages from Check #17. Else, default to error messages from Check #1.

Check #18a:
Allow for the usage of pointers to functions.
The grammar rule that defines a function pointer type is under the Type rule:
Type ::= ...  
| T_FUCNPTR T_COLON ReturnType OptRef T_LPAREN OptParamList T_RPAREN

An example of this is shown here:

typedef funcptr : int (int x, int y) MYPTRALIAS;
MYPTRALIAS myPtr1, myPtr2;

function : int addition(int x, int y) {
    return x + y;
}

type function : int subtraction(int x, int y) {
    return x - y;
}

function : int main() {
    if (myPtr1 == nullptr) {
        myPtr1 = addition;
    }
    cout << myPtr1(4, 6) << endl;
    myPtr2 = subtraction;
    cout << myPtr2(5, 2) << endl;
    myPtr2 = myPtr1;
    cout << myPtr2(5, 2) << endl;
    myPtr2 = nullptr;
    return 0;
}

The following items need to be checked:

- An assignment to a function pointer requires that the function pointer's type (formal parameters, including whether pass-by-reference (with an &)) or pass-by-value (default), and the return type) match exactly to the function prototype of the function trying to be assigned to it. Basically, check to see that the parameter list and return type are both identical in type and reference status. The parameter names (e.g. x, y) do not need to match. Note that even though void is not equivalent to itself, funcptrs of void type should be assignable to each other (assuming everything else matches, of course).
- Function pointers can be used like regular pointers, but are their own distinct type. You can compare function pointers with nullptr and assign them to one another or assign nullptr into them. Comparisons of function pointers to one another will not be tested.
- Additionally, you can assign the name of an actual function to a function pointer.

Note: you will need to modify some of the checking in the parser to not report an error when you try to do a function call using a function pointer. Currently, the code will report that function pointer is “not a function”. You will need to allow function pointers to go through and do the checking like any other function call.

There are no new error messages for this check. Issues with assignability should be handled using the messages from Check #3. Issues with equality/inequality should be handled by Check
#17.
The type (%T) you should display in any error messages for function pointers depends on the manner in which the pointer variable was defined. If the type was done using a typedef, simply use the typedef name. If the function pointer type was done directly within the variable declaration or if you are identifying an actual function as a function pointer, the following applies. For example, to print the type of function `addition`, it should be in the following format:

```
funcptr : int (int x, int y)
```

Notice that the parameter list is expanded and each parameter is printed with its corresponding type. Specifically, we want:

- One space after each punctuation character (except for parentheses and &);
- One space between each adjacent pair of words;
- The `funcptr` keyword, space, colon, space, return type, space, the & modifier if the function is return-by-reference (followed by space), open parenthesis (with no space after it), parameter list (if any), and close parenthesis (with no space before it);
- Each parameter declared printed separately (include the & modifier directly in front of the parameter name if specified in the declaration);
- All parameters in original declaration order.

Example of such formatting is shown below:

```
funcptr : int* (float*** x, bool* &y)
funcptr : int* & ()
```

Furthermore, if the error occurs on a variable defined with a typedef instead of the full type declaration, the name of the typedef is used instead, similar to the rest of the project. So, in the above example, if `myPtr2` had an error, the type would be `MYPTRALIAS`.

**Check #18b:**

Extend check #8 to detect an illegal initialization of function pointers

For function pointers, they can be initialized to nullptr, some other function pointer, or to the name of an actual function.

- **Constant function pointers will not be tested.**

Use the same error messages from check #8. Here are some examples:

```
function : int foo() { return 0; }
funcptr : int() fp1 = foo; // variable declaration with init
funcptr : int() fp2; // variable declaration without init
```
Check #19:
Implement "sizeof" for type and variable/constant objects. An error should be generated if the operand is not a type, or if the operand is not addressable.

The result of sizeof should be a constant R-value of int type with the proper size of the object. An example is provided below:

typedef float FOO;
structdef MS {
    int a, b;
};
int x;
const float y = 55.5;
bool[4] z;
MS t;

function : void foo(bool[4] &p1, bool* p2) {
    x = sizeof(p1);  // should be 16
    x = sizeof(p2);  // should be 4
}

function : void main() {
    foo(z, z);
    x = sizeof(FOO); // should be 4
    x = sizeof(MS);  // should be 8
    x = sizeof(float**); // should be 4
    x = sizeof(int[3]); // should be 12
    x = sizeof(x);    // should be 4
    x = sizeof(y);    // should be 4
    x = sizeof(z);    // should be 16
    x = sizeof(t);    // should be 8
}

Note: For the purposes of this assignment, the size of int, float, bool, and pointers is 4 bytes (this makes your lives much easier). **We won't test sizeof with function pointers.**

Check #20:
Allow for the usage of type casts, creating a new object type as specified. Below is an example of valid type casts:

typedef float FOO;
typedef FOO* FPTR;
typedef int* IPTR;
int x;
FOO y;
FPTR z;
int* intPtr;
function : int main() {
    x = (int) y;
    x = (int) (x + 4.9);
    y = (FOO) (int) (65.3);
    intPtr = (IPTR) z;
    return 0;
}

The only types that are acceptable for a cast are basic types (int, float, and bool), aliases to those types, and pointers (to any type). Entire structs and arrays cannot be cast, but individual elements within each can be cast if of the types listed above. Function pointers cannot be cast.

Furthermore, the result of a type cast produces an R-value. We will test having type casts on the left-hand side of an assignment statement, as well as within other places where an L-value is required (for example, passing a type casted variable to a pass-by-reference parameter should produce the appropriate L-value error, and passing a type casted variable to the address-of operator should produce the appropriate not addressable error).

Your type casts must also incorporate constant folding if the operand is a constant. In other words, if the source of the type cast is a constant, the result will be a new properly converted constant object. See the example below:

    const bool x = true;
    int[10] y;
    int z;
    function : int main() {
        z = y[(int) x];  // the index into the array will be 1
    }

Here are some casting rules for conversions of constants:

    bool --> int, float OR pointer: If true then 1, false then 0.
    int, float OR pointer --> bool: If ==0 then false, !=0 then true.
    float --> int OR pointer: Drop any decimals (3.99 becomes 3).
    int OR pointer --> float: Converted to FP pattern (42 becomes 42.00).
    int <---> pointer: No change in value.

We have provided one error message (error20_Cast) to cover cases of invalid type casts (i.e., casting an array... type into something).

Check #21:

Allow for the address-of (&) operator. The address-of operator is only valid on something that is addressable (hopefully, this is not surprising!). The resulting type will be a pointer to the original object’s type. The resulting type is no longer addressable or modifiable (it becomes an R-value), and should be an ExprSTO. If an address-of result is dereferenced, it will produce the original object’s type and become addressable again. Furthermore, it will become a modifiable L-value, even if the original object was not. An example is provided below:
int x, y;
int *z;
const int w = 77;
z = &x;     // &x in this example is simply an R-val
*x = y;    // &x is essentially just x, so OK.
*y = &x;   // The * reverses the &x, making it a modifiable L-val
*y = w;    // The * reverses the &w, making it a modifiable L-val,
    // even though w was originally a constant
*z = z;    // Error, result of address-of is not a modifiable L-val

Remember that the name of a function in an expression (without the parentheses) returns a
constant function pointer. The address-of operator cannot be used on a function name to
generate a pointer to a function pointer, since the function name is an R-value. Instead, you can
store the function name into a function pointer variable, and then take the address of that (as
shown below). This particular case should not require any additional effort, since the
generalized actions for address-of and dereference should produce the desired effect. An
example is provided below:

function : int foo() { return 0; }
typedef funcptr : int() MYFP;
MYFP MyFuncPtr;
MyFuncPtr = foo;
MyFuncPtr();   // this will be a function call to foo!

MYFP * MyFuncPtrPtr;
MyFuncPtrPtr = &foo;  // Should be an error, since foo is an R-value
MyFuncPtrPtr = &MyFuncPtr;  // OK
(*MyFuncPtrPtr)();   // this will be a function call to foo!

Check #22 Extra Credit (5%)

● Detect an illegal overloaded function definition:
A function definition is an illegal overload if there exists a previous definition with the same
name and an equal number of parameters of equivalent types. Note that for overloading
purposes, pass-by-value and pass-by-reference parameters count as having the same type.
Also note that return type and parameter names do not count in the function signature.

function : float foo(float &x) { ... }
function : int foo(float x) { ... }

is illegal, since they both have the same function name and one parameter of equivalent type.

● Detect an illegal overloaded call:
A call to an overloaded function is illegal if no overloaded instance is legal (i.e. a call for which
there is no exact match).

For overloaded function calls, there will be no automatic type promotion of argument types.
Promotion should occur normally for non-overloaded functions.
In the above example, `foo(1)` is illegal since the integer 1 will not be promoted into a float because `foo` is overloaded.

For function calls to functions with no overloading (i.e., only one function with that name), rely on the `error5n_Call` message. For overloaded functions, rely instead on the `error22_Illegal` message if the parameter counts don't match up.