CSE 131 – Compiler Construction
Fall 2015

Project #1 -- Semantic Analysis

Due Date: Sunday, October 25th, 2015 @ 11:59pm

IMPORTANT: You are responsible for having read and understood all items in this document.

Contents

Introduction 4
Foreword 4
Partners 4
Posting of Solutions 4
Disclaimer 4
Beginning your Project 5
Background 5
Running the Project 5
Turning in your Project 5
Handling Error Messages 6
Error Messages 6
Multiple and Cascading Errors 6
Printing Types in Errors 7
What we are NOT testing or using in test cases **(WNBT)** 8
The Assignment 9
Assignable vs. Equivalent 9
Modifiable L-Value vs. Non-Modifiable L-Value vs. R-Value 10
Equivalence Rules 10
Phase 0 (1st Week) 12
Phase 1 (~1 week) 13
Check #0 Scope resolution operator 13
Check #1 Binary operators 13
Check #2 Increment/decrement operators 14
Check #3a Illegal assignment 14
Check #3b Assignment type conflict 15
Check #4 If/while type conflict 15
Check #5 Function calls 15
Check #6a Return statement without expression 16
Check #6b Return statement type conflict 16
Check #6c Missing return statement 16
Check #7 Exit statement 16

Phase II (~1 Week) 17
Check #8a Constant/variable initialization (incl. constant folding) 17
Check #8b Constant/variable initialization using auto keyword 17
Check #8c Using named constants within expressions 17
Check #9a Overloaded function definition 18
Check #9b Overloaded function call 18
Check #10 Array declaration 19
Check #11 Array usage 20
Check #12a Foreach statement 21
Check #12b Break/continue statement 21
Check #13a Struct declaration, duplication identifiers 22
Check #13b Struct declaration, constructors/destructors 23
Check #14a Struct instantiation 24
Check #14b Struct usage 25
Check #14c Struct "this" keyword 25
Phase III (~1 week) 26
Check #15a Struct and pointer operands 26
Check #15b Pointers with array usage 26
Check #15c Pointer initialization 26
Check #16a New/delete statement 27
Check #16b New/delete statement for pointers to structs 27
Check #17 Extend operators to handle pointers 27
Check #18 Address-of operator 28
Check #19 Sizeof construct 28
Check #20 Type casts 29

Introduction

Foreword

This is a large and complex software project. It will take you many days to complete, so do not procrastinate and attempt to complete it the night before the due date. **IMPORTANT: There will be no extensions to the deadline and late submissions will absolutely not be accepted, regardless of things such as power failures, computer crashes, personal issues, etc.** It is your responsibility to start on this project early and turn-in your code often to ensure you do not miss the deadline. You can turn-in your code as many times as you want.

Partners

You are encouraged to work in teams of 2, although you may also work individually. While this is a large project, it certainly can be completed successfully by a single individual (and many of the best scores in the past years have been by individuals). **IMPORTANT: Keep in mind that if you do choose to work with a partner, you are both still responsible for the entire project, even if your partner decides to drop the class or for whatever reason no longer wants to work with you.** If you decide to work with a partner, it is advisable that you communicate with them at minimum every other day, so you both are kept up-to-date and can ensure progress is being made. I cannot count the number of times I've seen partners decide to divide the project up and assume the other is working diligently, only to find out a couple days before the deadline that their partner has done nothing and has just dropped the class. Don't put yourself in that situation. Communicate frequently with your partner, merge your code often, and if possible sit together and code together in the same room (or even same terminal if you like pair programming). Additionally, many questions on the quizzes and exams will be based on the project implementation. It is in your best interest to understand your entire project code, even if half the code was written by your partner.

Posting of Solutions

IMPORTANT: You must not post online or otherwise distribute to others your solution to this programming project. This prohibition applies both during and after the quarter. This prohibition specifically covers posting any programming assignments to GitHub or similar sites except in a private repository. Students violating this policy will be subject to an academic integrity disciplinary hearing.

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.

Beginning your Project

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. You should complete the phases in chronological order.

Running the Project

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

```
make new
./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 file is just a simple script that runs your Java project.

If you want to use Eclipse to work on the project, see the instructions in build.xml included in the starter code. If you choose to work with Eclipse, **you must still ensure you can compile and run from ieng9 command line with the Makefile** to ensure that the turn-in process works correctly and we are able to successfully grade your project.

IMPORTANT: All testing/grading will be done on ieng9, so it is imperative that you validate your compiler works in that environment using the command line Makefile to build your project. Failure to do so will result in your project getting a score of 0.

Turning in your Project

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. You may want to use the 'm_debugMode' member in MyParser.java to control outputting any debug statements (when ./RC is executed with the '-debug' switch).

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:

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

Handling Error Messages

Error Messages

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

The error messages in ErrorMsg.java have the place holders:

- %O for operators
- %T for types
- %D for integer values
- %S for identifiers/object names

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.

IMPORTANT: All error messages should be printed to standard output (stdout), not standard error (stderr).

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
```

If you would like to see the line number within the file where the error(s) occurred, run your compiler with the '-debug' switch passed to './RC':

```
make new
./RC -debug someTestFile.rc
Error, "file.rc", line 11:
 error message
```

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.

   IMPORTANT: 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.

IMPORTANT: 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 basic types, arrays, pointers, and structs should obey the following guidelines.

1. Printing basic types - print:
   - integer types as "`int`"
   - boolean types as "`bool`"
   - floating point types as "`float`".
2. Printing array types

   Printed array types include the dimension size(s) in brackets without any spaces (e.g. "`int[10]`" or "`bool[2][5]`").
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). While the RC language does support arrays of pointer types, it does not support pointers to array types, so the latter case will never be printed. The type of the nullptr keyword should be printed as "`nullptr`".
4. Printing structs

   Struct definitions are only done via a structdef. The only way a variable can be of a struct type is via the usage of the struct's identifier in the declaration. Thus, struct types should be printed using the name (the identifier) of the structdef. So for this example

   ```
   structdef MYSTRUCT {
         float field1;
         int   field2;
   };
   MYSTRUCT y;
   ```

   the type of "y" should print "`MYSTRUCT`".
5. 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`".

What we are NOT testing or using in test cases (WNBT)

- 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).
- Redeclared or undeclared identifier errors (the starter code already handles these).
- Nested function declarations or structdefs inside functions.
- The `ReadStmt/cin` and `WriteStmt/cout` rules (we will use these in the next project).
- The `T_STR_LITERAL` string literal terminal (we will use this in the next project).
- The `static` keyword for variables/constants (we will use this in the next project).
- The `extern` keyword for functions/variables (we will use this in the next project).
- 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)

```
int arr1[6];
int arr2[6];
structdef MS {
  int arr3[6];
};
function : int main() {
  arr2 = arr1;    // this will be tested and should generate an error
  MS s;
  s.arr3 = arr2;  // WNBT. s.arr3 is an expression of array type
  int arr4[2][6];
  arr4[1] = arr1; // WNBT. arr4[1] is an expression of array type
  return 0;
}
```

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

```
int myArrayOfArray[4][6];

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

- Passing arrays by value to functions (no value parameters of array type will be used).
- Functions returning arrays (no value or reference return of array type will be used).
- Address-of on expressions of array type (including directly on array names).
- Array declarations with initialization expression (i.e. `int x[5] = Expr;`).
- Using the name of a function in expressions other than during a function call (i.e. no function pointers will be used).
- Passing expressions containing errors as an argument to a function call.

```
function : void foo(int x) { ... }
function : void main() {
  foo(2+true); // WNBT
  foo(true);   // Will be tested, type mismatch
}
```

The Assignment

Assignable vs. Equivalent

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 coercion is promoting an integer to a float.

The following table shows some examples:

Term	**Types**
Equivalent	int <-> int float <-> float bool <-> bool int** <-> int** Type[5] <-> Type[5]
Assignable	Everything in *equivalent* int -> float (only in one direction)

When dealing with composite types like pointers and arrays, you need to compare that the base types are equivalent (should be done recursively for each level). For pointers you need to compare each level for equivalence until you reach the lowest/base type. For arrays, you similarly need to compare the dimension at the current level and if they match, check if the element types are equivalent (and in the case of arrays of arrays, you would repeat this until you get to the lowest/base type).

For example:

```
int * iptr1;
int * iptr2;
int ** iptr3;
float * fptr1;
float * fptr2;
float *** fptr3;
iptr1 = iptr2;     // ok
iptr1 = iptr3;     // generate error since base types are not equivalent
fptr1 = iptr1;     // generate error since base types are not equivalent
**fptr3 = fptr1;   // ok
```

IMPORTANT: The keyword/type `nullptr` is equivalent to any pointer type.

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

Modifiable L-Value vs. Non-Modifiable L-Value vs. R-Value

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:

Term	**Definition**	**Examples**
Modifiable L-value	Addressable and modifiable	• Variables • The results of expressions with the operands `* . -> []` (Note: array names are never modifiable lvals, even if accessed using `.` or `->`) • Function call expressions where the function returns by reference
Non-modifiable L-value	Addressable, but not modifiable	• Declared constants (e.g. `const int x = 5`) • The name of an array (array identifier)
R-value	Neither addressable nor modifiable	• The results from arithmetic operations (e.g. x+y) • Constant literals (e.g. 123) • The "this" and "nullptr" keywords • Results of address-of, sizeof, and type casts

Equivalence Rules

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

- All types (except structs) use **structural equivalence.**
- All structdefs use **strict name equivalence.**
- Struct-level operations (e.g. assignment, equality, and inequality) use **strict name equivalence**. All structs are defined with the `structdef` keyword.
- Struct identifiers are **modifiable L-values**.

Here is an example illustrating some of these points:

```
int i;
int m;
float r;

structdef REC1 { float a; };
structdef REC2 { float a; };
REC1 r1;
REC2 r2;
REC1 r3;

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

function : int f(REC1 &a) { /* stuff */ }
function : int g(float &a[5]) { /* stuff */ }

int* p1;
float* p2;
REC1* p3;
REC2* p4;
REC1* p5;

function : int main() {
   i = m;   // okay, assignable (both int)
   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/strict name equivalent

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

   a1 = a1; // error, array identifiers are non-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, strict name equiv & struct variables are mod L-vals

   p1 = p1; // okay, structurally equivalent
   p2 = p1; // error, not structurally equivalent
            // Error Message: error3b_Assign
   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;
}
```

Note that the time periods for the various phases are estimates to help you plan your time -- they are not due dates.

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 OptCtorCall 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 * -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. `-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

    Add parsing and conversion support to handle integer literals that are hexadecimal (`0x42` or `0X42`), and octal (`042`). This will be tested extensively.

Phase 1 (~1 week)

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

Check #0 Scope resolution operator

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:

```
int myX;
function : void foo(){
      int x;
      int i = 5 + ::x;   // Error, cannot find x in the global scope
      int j = 5 + ::myX; // No error
}
```

Check #1 Binary operators

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 right. 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 Increment/decrement operators

Detect a *type conflict in a pre/post increment/decrement* -- that is, for expressions like

```
y = x++;
w = x++ + --y;
```

an error should be generated if

- the type of the operand to the increment or decrement is not equivalent to 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 Illegal assignment

Detect an *illegal assignment* -- that is, an assignment of the form

```
myA = Expr
```

where myA is **not** a modifiable L-value. Some examples (most of which are implemented later in this project) include results of arithmetic expressions (e.g. (x+y) = 10;), function calls that return by value (e.g. foo() = 17;), constants (literals or declared), array designators, results of the address-of operation, results of the sizeof 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 structdef name is already caught by the starter code as a syntax error and does not need to be modified (we won't test syntax errors).

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 Assignment type conflict

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:

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

Check #4 If/while type conflict

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

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

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

where the type of `Expr` is not *equivalent* to bool.

Check #5 Function calls

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***.

IMPORTANT: 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:

```
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.
}
```

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 and struct-member functions will be tested in the later checks.

Check #6a Return statement without expression

Detect an *illegal return statement* where no *Expr* is specified and the return type is not void. Example:

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

Check #6b Return statement type conflict

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 return types.

Check #6c Missing return statement

Detect a *missing return statement*.

For functions not declared with void return type, at least one return statement (legal or illegal) must appear at the top level (i.e. not within a code block or an *if*, *while,* or *foreach* statement).

Check #7 Exit statement

Detect an *illegal exit statement* -- that is, statements of the form

```
exit(Expr);
```

where *Expr* is not *assignable* to an int.

Phase II (~1 Week)

Initializations, constant folding, function overloading, arrays, foreach, break/continue, and structs, in addition to Phase I tasks.

Check #8a Constant/variable initialization (incl. constant folding)

Detect an *illegal constant/variable initialization*

```
const Type c = ConstExpr;
Type x = Expr;
```

For this example, an 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 know 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 #8b Constant/variable initialization using auto keyword

The C++11 `auto` keyword allows the compiler to infer the type of a constant/variable from the type of the initializer. Instead of declaring a constant/variable with a specific type, you can declare it with `auto`:

```
int a;
auto b = a;             // b is of type int
auto c = b * 1.0;       // c is of type float
const auto d = true;    // d is of type bool
```

Only initialization expressions of basic types (`int`, `float`, and `bool`) will be tested. This will be extended in Check #15c to also handle initialization expressions of pointers (to any type, excluding `nullptr`).

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 #8c Using named constants within expressions

Extend all previous checks to allow for named constant operands (example: `const int Zero = 0`)

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:

```
int a;
int b;
const int c = 2 + 3 * 0 - 1;
const int d = c * 7;
const float e = (d + c) * c;
function : int main() {
    a = (b + c + d);
    return (a % c);
}
```

Check #9a Overloaded function definition

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 matter 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.

Note: Currently the starter code will emit a "redeclared identifier" error when you declare more than one function with the same name. You will need to modify the logic in DoFuncDecl_1 to only emit "redeclared identifier" if the existing identifier is not a function (i.e. a variable that has the same name as the function). You will then need to check if an illegal overloaded function definition occurs once you parse the parameters. This checking and outputting of the `error9_Decl` error message can be done in DoFormalParams.

Check #9b Overloaded function call

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/coercion of argument types. Promotion/coercion should occur normally for non-overloaded functions.

```
function : float foo(float x) { ... }
function : float foo(float x, float y, float z) { ... }
function : int main() {
    foo(1);             // error9_Illegal
    return 0;
}
```

In the above example, `foo(1)` is illegal since the integer 1 will not be promoted into a float because function foo is overloaded.

For function calls to functions with no overloading (i.e., only one function with that name), rely on the Check #5 error messages. For overloaded functions, rely instead on the `error9_Illegal` message if no exact semantically correct match is found (including cases where one or more parameters are pass-by-reference and the corresponding argument is not a modifiable L-value). Only one error9_Illegal message should be displayed for an erroneous overloaded function call, regardless of how many of the arguments had issues.

Note: Overloading of struct-member functions and having array, pointer, and struct parameters will be tested in later checks.

Check #10 Array declaration

Detect an *illegal array declaration*.

Given a type declaration such as

```
Type Id [Index];
```

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: You can have any number of arrays of arrays by listing additional indexes:

```
float a[10];
float b[10];
float c[2][10];          // array of array (2 rows, 10 columns)
bool  d[5][4][3][2][1];  // crazy!
```

The grammar defines the non-terminal ArrayList which will collect all the `[Index]` components. As a suggested design choice, you can capture this list of index sizes inside a Vector object that will eventually bubble-up to the VarDecl rule. At that point you can create the corresponding recursive Type object chain using the knowledge of the underlying base type (**`Type`** in this case). This recursive Type object chain will be extremely helpful when you implement the next check for accessing different levels of the array or array using the [] operator.

IMPORTANT: If there is an error with multiple index expressions in a single array declaration, only output the first error encountered in the order that the expressions are defined (i.e. from left to right).

Note: Initialization expressions WNBT with array declarations (e.g. float a[10] = ... ;)

Check #11 Array usage

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 2 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]).

```
const int cc = 1;

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

c[1][9] = a[9];       // no error, within bounds
b[b[b[b[2] - cc] + cc] - 1] = c[1][9];
a[2] = d[0] + d[99999];
```

Note: Arrays will only be **passed-by-reference** to array 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 equivalent reference parameters as arguments*.

Check #12a Foreach statement

Detect an *illegal foreach loop statement*

Given a *foreach loop statement* of the following form

```
foreach (IterationVar : Expr) { ... }
```

An error should be generated if

- The type of `Expr` is not an array type;
- The `IterationVar` is declared as **value** (default) and the `Expr`'s element type is not ***assignable*** to the `InterationVar` type;
- The `IterationVar` is declared as **reference** (using the &) and the `Expr`'s element type is not ***equivalent*** to the `InterationVar` type;

Consider the following examples:

```
int     a;
int     b[2];
float   c[3];
float * d[4];
float   e[3][4];

foreach (float  x : a) { ... } // Error, 'a' is not an array type
foreach (bool   x : b) { ... } // Error, int not assignable to bool
foreach (float  x : b) { ... } // OK, int assignable to float
foreach (float &x : b) { ... } // Error, int not equivalent to float
foreach (float*&x : d) { ... } // OK, float* equivalent to float*
foreach (float  x : d) { ... } // Error, float* not assignable to float
foreach (float  x : e) { ... } // Error, float[4] not assignable to float
```

Check #12b Break/continue statement

Detect an *illegal break or continue statement*

```
break;
```

or

```
continue;
```

Errors should be generated if either statement is not within the body of a `while` or `foreach` loop.

Check #13a Struct declaration, duplication identifiers

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:

```
structdef MYS {
      int x;
      int y;
      int x;      // check13a_Struct dup id x, error #1
      int z;
      int x;      // check13a_Struct dup id x, error #2
      MYS * next;       // recursive field – no error – will be tested
      function : void y() {}  // check13a_Struct dup id y, error #3
      function : void f( int &x )   // No error with x (inner scope)
      {
         x = x + 1;
      }
      function : void foo() {}
      function : void foo() {}  // error9_Decl dup overload, error #4
      function : void bar() {}
      function : void bar(int x) {}  // OK, since legal overload
};
```

IMPORTANT: We will always declare all struct-member variables prior to any struct-member functions (e.g. no struct-member variable definitions will be placed after struct-member functions within the structdef).

IMPORTANT: struct-member function names are in the same namespace as the other variable identifiers within the struct. A function name that is the same as a variable within the struct should result in a duplicate field error as well. However, overloaded struct-member functions are legal and follow the same rules as Check #9. Any struct-member functions referenced by other struct-member functions will be declared prior to such references (since we have a one-pass compiler).

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

Below is a summary of the appropriate error message that should occur for struct-member variables and functions depending on the situation:

- You will first encounter the struct-member variables.
  - If any variable names are duplicated, output a `check13a_Struct` error message for each duplicate variable instance.
- After all the struct-member variables, you encounter the struct-member functions.
  - First check if the struct-member function has the same name as a struct-member variable. If so, output a `check13a_Struct` error message for each duplicate instance.
  - If the struct-member function has a name that is different that all the struct-member variables, then you need to check if that struct-member function has the same name as another struct-member function and follow the rules of Check #9 to output a `error9_Decl` error message for each duplicate overload.

Check #13b Struct declaration, constructors/destructors

Detect an *illegal struct declaration* -- invalid constructor/destructor definitions.

Struct declarations may contain constructors and/or destructors. The key differences between a regular struct-member function and a struct constructor/destructor is that the latter has no 'function :' prefix, no return type or void keyword (although you should implicitly consider them as having void return type to allow for 'return;' statements), and the name of the function matches the name of the structdef itself (and thus needs to be all uppercase to be syntactically correct). *Constructors can be defined with a formal parameter list and be overloaded, similar to struct-member functions. Destructors are prefixed with a '~' character, and have no parameter list (and as such any attempt to overload destructors will result in an error).*

Errors should be generated for each constructor/destructor definition if

- The name of the constructor or destructor does not match the name of the structdef;
- More than one constructor is defined and results in an illegal overload (follow the same rules as in Check #9 to determine if the overload is illegal; use message `error9_Decl`);
- More than one destructor is defined (since destructors have no parameter list, these will always be illegal overloads - use error message `error9_Decl`).

For example:

```
structdef MYS {
      int x;
      MYS() {}          // ctor – no error
      MYS(int x) {}     // OK, legal overload – no error
      MYS(float x) {}   // OK, legal overload – no error
      ~MYS() {}         // dtor – no error
      ~MYS(int x) {}    // syntax error (no params in dtor) – WNBT
      FOO() {}          // error13b_Ctor - name does not match struct
      ~BAR() {}         // error13b_Dtor - name does not match struct
      MYS() {}          // error9_Decl – dup overload
      MYS(int y) {}     // error9_Decl – dup overload
      ~MYS() {}         // error9_Decl – dup overload
};
```

Note: when outputting the name of the destructor in error messages, include the ~ character directly before the destructor name (without any space between them).

Note: The grammar requires any constructors and destructors be defined after struct-member variables and prior to any struct-member functions. Thus, constructors/destructors will be able to access member variables (using the "this" keyword described in check #14c below) but not be able to call member functions (since our compile is one-pass). Additionally, there is no way for a structdef's constructor to be explicitly called from within that structdef (i.e. you cannot chain constructors like what is commonly done in Java).

At the end of the parsing ctors/dtors and before parsing member functions, if no constructors were defined you should automatically create and insert a default constructor definition (with no parameters) into your structdef type.

Check #14a Struct instantiation

Detect an *illegal struct instantiation*.

Struct variables can be instantiated in both global and local scopes. Similar to other variables, they can be defined simply by specifying the struct's type and the variable name. This will result in an implicit call to the default constructor if one exists (i.e. the constructor with no parameters). Alternatively, you can explicitly call the constructor by appending "`: ( args )`" after the variable name. Some examples of such instantiations are shown in the code below.

Errors should be generated if the struct instantiation does not properly match a corresponding constructor in the structdef:

- If the structdef defines only a single constructor (i.e. no overloading), follow the rules of Check #5 to verify a match or output the appropriate error message from Check #5.
- If the structdef defines more than one constructor (i.e. overloaded), follow the rules of Check #9b to verify a match or output the `error9_Illegal` error message.

Note: Arrays of structs can be instantiated, and constructor calling should be similarly checked.

Examples of such cases are provided below:

```
structdef MYS_A {
    int x;
};              // Your compiler will implicitly add default ctor in this case
structdef MYS_B {
    int x;
    MYS_B(float x) {}
};
structdef MYS_C {
    int x;
    MYS_C() {}                  // Explicit default ctor defined
    MYS_C(int & x) {}
    MYS_C(int x, float y) {}
};

// Global instantiations
MYS_A a1;                       // OK - matches default ctor
MYS_A a2 : ();                  // OK - matches default ctor
MYS_A a3 : (1, 2);              // error5n_Call - implicit 2 arg call, 0 param
MYS_A a4[5][1];                 // OK - matches default ctor
MYS_A a5[5] : ();               // OK - matches default ctor

function : void main() {
    // Local instantiations
    MYS_B b1;                   // error5n_Call - implicit 0 arg call, 1 param
    MYS_B b2 : ();              // error5n_Call - explicit 0 arg call, 1 param
    MYS_B b3 : (1);             // OK - int coerced to float (no overloading)
    MYS_B b4 : (false);         // error5a_Call - not assignable
    MYS_B b5[7];                // error5n_Call - implicit 0 arg call, 1 param
    MYS_B b6[7] : (2.0);        // OK - matches ctor
    MYS_C c1;                   // OK - implicit call to default ctor in MYS_C
    MYS_C c2 : ();              // OK - explicit call to default ctor in MYS_C
    MYS_C c3 : (1);             // error9_Illegal - no exact overload match
    MYS_C c4 : (1, 2.2);        // OK - exact match to overloaded ctor in MYS_C
    MYS_C c5 : (1, 5);          // error9_Illegal - no exact overload match
    MYS_C c6 : (1, 2, 3);       // error9_Illegal - no exact overload match
 }
```

Check #14b Struct usage

Detect an *illegal struct usage*.

Given a designator such as

```
MyStruct.SomeField
MyStruct.SomeFunc()
```

an error should be generated if

- the type of **`MyStruct`** is not a struct type
- the type of **`MyStruct`** has no field named **`SomeField`** or function named **`SomeFunc`**

Note: the same function calling requirements of Check #5 and Check #9b apply to struct-member functions.

Check #14c Struct "this" keyword

Handle the "`this`" keyword within struct-member functions

The "`this`" keyword is an R-value of the same type as the enveloping structdef, and is needed to access any struct fields and member functions from within a struct-member function or constructor/destructor. Below is an example:

```
structdef MYS {
      int x;
      int y;
      MYS() {                  // Ctor
         this.x = 0;
         this.y = 0;
         this = nullptr;       // Error, not modifiable l-val
         this.x = this;        // Error, MYS not assignable to int
      }
      MYS(int x, int y) {      // Ctor
         this.x = x;
         this.y = y;
      }
      function : void foo() {
         this.x = 8;
         this.y = this.x;
         this.foo();           // Legal, recursive call
      }
      function : void bar() {
         this.foo();           // Legal, calling other struct function
      }
};
```

Similar to check #14b above, you need to verify that the field/function specified after the "." exists in the scope of the current struct. The error message would be *error14c_StructExpThis*.

IMPORTANT: The "`this`" keyword will only be tested inside struct-member functions and constructors/destructors. Also, the "`this`" keyword is a strict requirement to access fields/functions belonging to the struct. Without "`this`", only the local and global scopes are checked for the identifier, and the struct scope is bypassed.

Phase III (~1 week)

Pointers, address-of, sizeof, type casts.

Check #15a Struct and pointer operands

Extend all previous checks to test for operands consisting of

- Structs and pointers, and pointer dereferences
- Functions with pointers and struct arguments and/or return types

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;
(*ptr).x = y;
ptr->x = w;
*ptr2 = z;
```

An error should be generated if

- The keyword `nullptr` is dereferenced using the *, ->, or [] operators.
- 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 can be passed/returned-by-reference or passed/returned-by value to functions.

Check #15b Pointers with array usage

Extend check #11 to allow pointer designators for the [] array dereference operator. Remember that the array bounds checks should not occur for pointers, since the size is unknown.

```
int * x;
x[99] = x[-40] + x[*x];       // No errors
```

Check #15c Pointer initialization

Extend check #8 to detect an *illegal initialization of pointers.*

Pointers variables can be initialized to nullptr, some other pointer, or the result of an address-of operation (discussed in check #18). Use the same error messages from check #8. Support the `auto` keyword for pointer initialization expressions to any type (excluding `nullptr`). Some examples:

```
int x;
int* r = &x;
auto s = r;
```

Check #16a New/delete statement

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 a 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 #16b New/delete statement for pointers to structs

Detect an *illegal new statement for a pointer to a struct.*

For a statement of the form:

```
new x : ( args );
```

an error should be generated if

- `x` is not a pointer to a struct type

Additionally, in the case of both '`new x;`' and '`new x : ( args );`', if `x` is a pointer to a struct, then the same rules and error messages specified in Check #14a should be used to validate that the struct type has a valid matching constructor.

Note: We will not test calling `new` or `delete` on a recursive struct pointer from within said struct except for within that struct's member functions (i.e. after the struct's constructors/destructors are defined, since those come before member functions). Pointers to other structs (non-recursive) or non-struct types can have `new` and `delete` called on them from within both struct-member functions and struct constructors/destructors as would normally be expected.

Check #17 Extend operators to handle pointers

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`.
- For the pre/post increment/decrement (see Check #2), allow for operands of type equivalent to a pointer type. 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 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 #18 Address-of operator

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;
int y;
int *z;
const int w = 77;
z = &x;       // &x in this example is simply an R-val
&x = nullptr; // Error, since not a modifiable L-val
y = *&x;      // *&x is essentially just x, so OK.
*&x = y;      // The * reverses the &x, making it a modifiable L-val
*&w = y;      // 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
```

Check #19 Sizeof construct

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:

```
structdef MS {
      int a;
      int b;
      function : int doo() { return 0; }  // has no effect on size!
};
int x;
const float y = 55.5;
bool z[4];
MS t;
function : void foo(bool &p1[4], bool* p2) {
      x = sizeof(p1);          // should be 16
      x = sizeof(p2);          // should be 4
}
function : void main() {
      foo(z, &z[0]);
      x = sizeof(float);       // 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
}
```

IMPORTANT: For the purposes of this assignment, the size of int, float, bool, and pointers is 4 bytes (this makes your lives much easier).

Check #20 Type casts

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

```
int x;
float y;
float* z;
int* intPtr;

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

The only types that are acceptable for being cast are basic types (`int`, `float`, and `bool`) and pointers (to any type, excluding `nullptr`). Entire structs and arrays cannot be cast, but individual elements within each can be cast if of the types listed above. The keyword `nullptr` cannot be cast and should also result in an error.

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 and the target type is non-pointer. 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 y[10];
int z;
function : void main() {
 z = y[(int) x];  // the index into the array will be 1
}
```

Here are some casting rules for conversions of constants:

```
bool --> int OR float: If true then 1, false then 0.
int OR float --> bool: If ==0 then false, !=0 then true.
float --> int: Drop any decimals (3.99 becomes 3).
int --> float: Converted to FP pattern (42 becomes 42.00).
```

We have provided one error message (*error20_Cast*) to cover cases of invalid type casts.

IMPORTANT: When casting from float directly to bool, you should strictly compare if the float value is zero or non-zero to determine if false or true, respectively. You may see the conversion from float to bool result in true in cases where imprecision resulted in a value like 0.0000000001.