COSC 4785/5785
Compiler Construction I
Program 6, Fall 2021

1 Introduction

This is the final program for the Compiler Construction course. It will be due during Finals week on the date/time that the final exam is scheduled. There can be NO late submissions. You must make sure that everything actually works. Yes, there may be some errors, but they should not be serious. That all means you have to test your program thoroughly.

1. If you have any questions about any of this, you need to ask. Please do not make possibly unfounded assumptions and decide at the last minute that they might not be right.

2. This is the final programming assignment for the course. It will bring together all that we have discussed regarding lexical analysis, syntactic analysis, semantic analysis, and type checking. This document contains some additional rules that you will have to implement for your "compiler".

3. This all means that your current version of the compiler needs to be adapted to implement all the remaining type checking rules, such as validating method calls, array accesses and use.

4. Additional rules not apparent in the grammar:

   (a) There must be a single method named ***main***. There will only be one class which implements this method. ***main*** must have a return type of ***void*** or ***int***. It takes no arguments.

   (b) Ensure that ***this*** is only used at the beginning of a dotted name. It is a reference type to a class. What class depends on where it is used. If a method in class **foo** references ***this***, then within the scope of the method, ***this*** is a reference to **foo**.

   (c) Arithmetic operators all require **int** operands. This includes the operators $>, <, >=, <=$.

   (d) All relational operators return 1 if true and 0 if false. Of course the not (**!**) operator returns 0 if its operand is true and 1 if false. For the purposes of this small language, the conditional operators (&& and ||) only take integers as operands. The equality operator can compare any reference (class) type to **null** or to another reference type, otherwise it only works on integers (for ease of implementation).

(e) ***new***: ***new*** allocates memory for an object, calls its constructor and returns a reference to the new object. Then the type on the right side must match the type of the reference variable on the left side. Additionally the arguments (to the constructor) must match the types of the parameters.

For ***new*** statements involving arrays, you should understand that as long as the left-most bracket pairs (at least one) in the **new** expression contain integer expressions, the right-most (at *most* one) can be empty, to be "filled" later. Thus

```
int[][] a;
int[][][][] c;
a = new int[5][];

c = new int[10][2][9][];
```

are correct. But

```
int[][][][] c;

c = new int[10][][][];
```

is not correct.

(f) Array return types: This is the correct way to specify that a method returns a reference to an array

```
int [] foo
```

And of course, there can be multiple dimensions.

(g) ***null***: ***null*** is a reference type to a class or an array. Its type matches any reference or array type. in other words, every reference or array type may be thought of as a subtype of ***null***. It cannot be compared to an integer.

(h) ***read()***: ***read()*** is an operator that returns an integer. It takes no arguments.

(i) ***print()***: ***print()*** returns a ***void***. It takes as arguments, 0 or more integers (as in print(a,45,b,c);).

(j) Constructor methods cannot be called directly. They are only called when a ***new*** expression is executed. So a direct constructor call is an error.

(k) It is an error for a method having a return type of ***void*** or for a constructor to have a ***return*** statement that includes the optional expression.

(l) Methods do not have to have a return statement as the last line of the method, even if the method has a return type other than void. And they do not have to have any return statement at all. If you were doing code generation this

would mean that integer methods would by default return 0 and reference type methods would by default return ***null***

Better compilers than ours normally check to ensure that all execution paths in non-void methods have a return, but only issue warnings if some path is missing a return.

5. Type checking will be comprehensive. That is all the type checking concepts that have been discussed previously in the course will be implemented. As you have all the symbol tables working that should not be a problem.

2 Code Generation

Code generation will will be very simplified. There is none.

3 What to Turn In

This final program should really DO all of the things that the first 5 versions were supposed to accomplish. The output WILL correspond to that specified from Program 5. Those instructions are duplicated here.

3.1 Format of output

1. A set of error messages. These are same error messages you should have lovingly crafted in the previous two versions of the compiler. If the program has no errors, nothing should be printed for this part.

2. After having processed the entire input, you should NOT PRINT any of the information from the first four assignments. I do not want to see that. You should instead begin type checking. If there are type errors, please try to print out as much information as possible about the error and where it occurred. Preferable something like

```
invalid l-value, line 23: 5 = x + 6;
type mismatch in expression, line 28; x / y
```

I know that line numbers and the original input are a problem, just do the best you can.

3. Once the type checking is done, your program should *dump* the symbol tables. That is, output information that shows what the contents of the tables are and their relationship. We will do this as simply as possible. Basically, process any global table, printing out its entries, one per line. Then do an in order traversal of the rest of the symbol tables printing their content as you go. This means that each block (scope) will be printed, then the blocks contained in it, and so forth. Each scope should be offset two spaces from the parent scope, something like:

```
foo  class_type
  x int
  y int
  foo  method_type  null <- null
  something method_type int <- int x int
    a int
    b int
    c int

Bill class_type
  x int
  .
  .
  .
and so on
```

The two sections of the output should be distinct. That is the error output should occur first (as the input is processed), type errors next, and the symbol table dump should be last. The output from the previous assignments should be eliminated **except** for the errors with their corresponding line and column number.

3.2 Submission

1. Create a tar archive of the source files for you current version of the compiler. If you are using compression on the tar file, the CORRECT file extension is **.tgz**. Otherwise it is **.tar**. Submit that file on the WyoCourses assignment page.

2. Your source code should have comments (like your name, date, course at the beginning). These should identify the file and explain any non-obvious operations. Feel free to comment as necessary to help yourself but do not comment every line.

   Including a "readme" of some type is not necessary but if there are things you think I **need** to know about your code, feel free. I prefer **.txt** files for this.

   Include a VALID Makefile. The name of the executable should be **program6**.

   DO NOT *tar* DIRECTORIES. I only want files when I extract the archive.