CSE 5343, Programming Project 3: Semantic Checking for simpleC
Due Tuesday, February 13, 11:59 pm (30 points)

The goal of this project is to implement some simple semantic checking for the AST produced by Project 2. The checking will be used to (1) reject invalid programs, and (2) compute information that is needed for the next project. Create a directory `p3` under `proj`. Copy your Project 2 to `p3` and start from there. Do all work for Project 3 in `p3`. The starting point should be a call astRoot.check() inside class Compiler, after the parsing has completed successfully.

Goals

Symbol table. The information from declarations should be stored in a symbol table and then used to type check all expressions. Each declared variable should be recorded in the symbol table, together with its type. You need to decide how to represent types. For example, if the program contains declarations **double x; int y[10][20][5];** you need a representation of types *double* and *array(int,10,20,5)*. Then the symbol table can map the string *x* to type *double* and the string *y* to type *array(int,10,20,5)*. One simple way to achieve this is to map an id to the Decl AST node that declares it.

In the process of constructing the symbol table, perform the following semantic check, which is similar to a check done for real C programs.
Check 1: There cannot be multiple declarations for the same variable name
For example, **double x; int y; double x;** should trigger a checking error. In addition,
Check 2: Each dimension in an array declaration is a positive value
For example, declaration **int a[5][0][10];** should trigger an error.

Type checking for expressions. The most basic form of semantic checking is type checking. Perform type checking of all expressions in the program. As part of this process, record the type of each expression in a new Java field that you add to class Expr. Use the version of type checking that **allows widening conversions** from *int* to *double*, as discussed in class.
Check 3: Each binary expression should be type checked as discussed in class
One refinement: according to the C standard, the second operand of % must be of type **int** (Section 6.5.5). Similarly, the second operand of %= must be of type **int**. Your project should implement these refinements.

In addition, you need to handle multi-dimensional arrays, which were not discussed in class.
Check 4: The number of array dimensions in an array access expression must match the corresponding array declaration
For example, **int a[5][10][20]; a[1][2] = 3;** should trigger an error because the array access expression **a[1][2]** has two dimensions instead of three. In addition, as discussed in class, expressions for array indices should be of type *int* (this is part of Check 3). For example, this should trigger an error: **double x; int a[10][20]; a[2][x+1]=20;** because **x+1** is of type *double*.

Assignments. The left operand of an assignment operator =, +=, etc. cannot be an arbitrary expression. In general, it should be an expression that has an *l-value* – that is, it designates a location in memory and therefore can be on the left-hand side of an assignment. Expressions such as **3** and **x+y** do not have l-values.
Check 5: The left operand of an assignment operator must be a variable or an array expression

Statements. Every expression appearing in a statement must be type checked. Note that in our language (and in C) the conditional expressions in if-then, if-then-else, and loops can be of any type – for example, **double x; if (x) …** is allowed. In addition to checking all expressions, the expression in each return statement **return expr;** must be checked against the declared return type of the function. For example, **int f() { return 3.14; }** must trigger an error.
Check 6: The expression type in a return statement must match the return type of the function

Errors. In class Compiler, create a new exit code EXIT_SEMANTIC_CHECKING_ERROR with value 2. If the program violates any of the checks, call fatalError with this exit code. The test script will check this exit code, so please make sure your implementation uses it. The text message associated with the error should be something simple that describes which specific check was violated. Your code should call fatalError as soon as it detects a violation. If the program contains several type errors, only the earliest one will be detected and reported.

Temporary variables. If the program is successfully type checked, you need to perform additional setup work for the next project. Project 4 will need compiler-generated temporary variables. In Project 3, create such a variable for each binary expression *except for* binary expressions that use the assignment operator =. For example, if the input is
int f() { int a; int b; int c; int d; d = a+b+c; return d+1; }
you need to create 3 temporary variables since there are three + operators. The binary expression using the = operator does *not* need a temporary variable; do not create a temp for it. Name these variables **_t1**, **_t2**, … Assume that the input program does not use such names. Each temporary variable has the same type as the corresponding binary expression. During type checking, create such variables and add them to the symbol table. You can record each such variable in the corresponding expression AST node, which will be useful later in Project 4.

Temporary variables should also be created for array access expressions (class ArrayExpr). The type of the temporary should be the same as the type of array elements. For example, for
int f() { int a[10][20]; a[2][2] = 8; return a[2][2]; }
you should create two temporary variables of type **int**. Some of these variables may be redundant in Project 4 (will never be used), but that's OK – still create them in Project 3.

After type checking, use astRoot.print(System.out) inside class Compiler to print the program, but during printing, show the temporary variables in the list of declarations. For example, for the first program from above, the output could be
int f() { int a; int b; int c; int d; int _t2; int _t1; int _t3; d = a+b+c; return d+1; }
The order inside the list of declarations can be arbitrary: e.g., for this program, any permutation of these 7 variable declarations is fine. Implement this printing inside method Program.print.

The resulting output must be a compilable C program. In your testing, check this requirement using gcc -c my_output_prog.c

Testing

Your submission must work correctly on test program lpc.c provided on the web page. *You can expect that a substantial number of points in the grading will be related to this test case.* Make sure that your project produces a compilable and executable C program, as was done in Project 2.

Write many test cases and test your implementation with them. Submit at least 5 test cases with your submission. The test cases you submit will not affect your score for the project. Put them in the same location as the provided file t1.c and name them t2.c, …

Submission

After completing your project, do

```
cd p3
make clean
cd ..
tar -cvzf p3.tar.gz p3
```

Then submit `p3.tar.gz` in Carmen.

General rules (copied from the course syllabus)

Your submissions must be uploaded via Carmen by midnight on the due date. The projects must compile and run on **stdlinux**. Some students prefer to implement the projects on a different machine, and then port them to stdlinux. If you decide to use a different machine, it is entirely your responsibility to make the code compile and run correctly on stdlinux before the deadline. In the past many students have tried to port to stdlinux too close to the deadline, leading to last-minute problems and missed deadlines.

Projects should be done independently. General high-level discussion of projects with other students in the class is allowed, but **you must do all design, programming, testing, and debugging independently.** Projects that show excessive similarities will be taken as evidence of cheating and dealt with accordingly. Code plagiarism tools may be used to detect cheating. See the syllabus under "Academic Integrity".

The projects are due by 11:59 pm on the due day. You can submit up to 24 hours after the deadline; if you do so, your score will be reduced by 10%. **ONLY THE LAST SUBMITTED VERSION WILL BE CONSIDERED. Triple-check carefully that you have submitted the correct version. If you submit the wrong version of your code, and you get a low score (or zero score), I will NOT consider resubmissions – the original low/zero score will be assigned WITHOUT DISCUSSION.**

If you submit more than 24 hours after the deadline, the submission will not be accepted. NO EXCEPTIONS TO THIS RULE WILL BE CONSIDERED. NO REQUESTS FOR RESUBMISSION WILL BE CONSIDERED. MAKE SURE YOU SUBMIT THE CORRECT CODE VERSION.

Read the project description **very carefully, several times, start-to-end.** If you need any clarifications, contact me immediately (do **not** wait until the last minute). **Test extensively.**

Accommodations for sickness and other special circumstances will be made based on university guidelines. Please contact me **ahead of time** to arrange for such accommodations.