CS 445 – Spring 2023

Homework 4

Due on or before 12 Mar at 11:59 PM Pacific Time.

WARNING

Because the output of your program for this assignment will first be processed by an automatic comparison program before being examined by a human being, please follow formatting instructions/examples very carefully. The results your program produces will need to **look exactly like the examples**. Do not embellish with extra titles or other text such as "run complete" or "CS445 output" or even any extra spaces. Points will be deducted for this. (This is realistic. Most companies run test suites on their products and breaking the test suites is not looked upon well in industry.) The testing facility of the submit script will help you refine your software to the point that it is completely correct if you use it as intended.

1. The problem

In this assignment we will continue semantic analysis and error generation. We will also add I/O run-time library support.

1.1 New Compiler Messages

For this assignment your compiler will detect more semantic errors/warnings and emit messages for them. You will be graded on detecting all errors/warnings and emitting messages, including those that are from the last homework, so be sure that you complete the semantic analysis and error message generation that was included in the last homework. The list of all errors/warnings to detect and the messages that your compiler will emit for them is shown below:

```
"ERROR(%d): '%s' is a simple variable and cannot be called.\n"
"ERROR(%d): '%s' requires both operands be arrays or not but lhs is%s an array and
      rhs is%s an array.\n"
"ERROR(%d): '%s' requires operands of %s but lhs is of %s.\n"
"ERROR(%d): '%s' requires operands of %s but rhs is of %s.\n"
"ERROR(%d): '%s' requires operands of the same type but lhs is %s and rhs is %s.\n"
"ERROR(%d): Array '%s' should be indexed by type int but got %s.\n"
"ERROR(%d): Array index is the unindexed array '%s'.\n"
"ERROR(%d): Cannot have a break statement outside of loop.\n"
"ERROR(%d): Cannot index nonarray '%s'.\n"
"ERROR(%d): Cannot return an array.\n"
"ERROR(%d): Cannot use array as test condition in %s statement.\n"
"ERROR(%d): Cannot use array in position %d in range of for statement.\n"
"ERROR(%d): Cannot use function '%s' as a variable.\n"
"ERROR(%d): Expecting %s in parameter %i of call to '%s' declared on line %d but
      got %s.\n"
"ERROR(%d): Expecting %s in position %d in range of for statement but got %s.\n"
"ERROR(%d): Expecting Boolean test condition in %s statement but got %s.\n"
"ERROR(%d): Expecting array in parameter %i of call to '%s' declared on line %d.\n"
```

```
"ERROR(%d): Function '%s' at line %d is expecting no return value, but return has a
      value.\n"
"ERROR(%d): Function '%s' at line %d is expecting to return %s but return has no
      value.\n"
"ERROR(%d): Function '%s' at line %d is expecting to return %s but returns %s.\n"
"ERROR(%d): Initializer for variable '%s' is not a constant expression.\n"
"ERROR(%d): Initializer for variable '%s' of %s is of %s\n"
"ERROR(%d): Initializer for variable '%s' requires both operands be arrays or not
      but variable is%s an array and rhs is%s an array.\n"
"ERROR(%d): Not expecting array in parameter %i of call to '%s' declared on line
      %d.\n"
"ERROR(%d): Symbol '%s' is already declared at line %d.\n"
"ERROR(%d): Symbol '%s' is not declared.\n"
"ERROR(%d): The operation '%s' does not work with arrays.\n"
"ERROR(%d): The operation '%s' only works with arrays.\n"
"ERROR(%d): Too few parameters passed for function '%s' declared on line %d.\n"
"ERROR(%d): Too many parameters passed for function '%s' declared on line %d.\n"
"ERROR(%d): Unary '%s' requires an operand of %s but was given %s.\n"
"ERROR(LINKER): A function named 'main' with no parameters must be defined.\n");
"WARNING(%d): Expecting to return %s but function '%s' has no return statement.\n"
"WARNING(%d): The function '%s' seems not to be used.\n"
"WARNING(%d): The parameter '%s' seems not to be used.\n"
"WARNING(%d): The variable '%s' seems not to be used.\n"
"WARNING(%d): Variable '%s' may be uninitialized when used here.\n"
```

Note that the majority of these messages were contained in homework 3, so if you correctly completed that homework you have less to accomplish for this homework. As with the last homework, your goal is to replicate the output shown in the example files that are contained in the course website. You should continue to keep a count of the number of errors and warnings that were encountered while processing a given input.

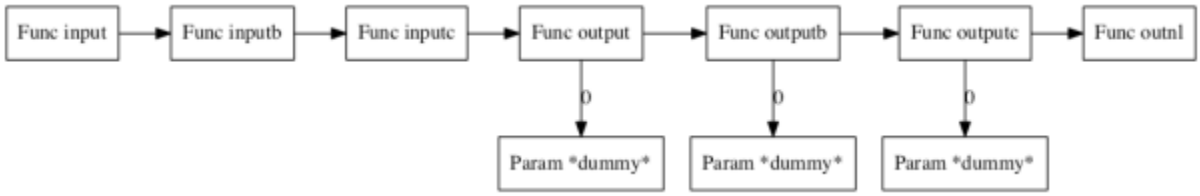
1.2 Adding I/O Runtime Support

For this assignment you are to add nascent support for I/O runtime library routines that will be used in C- programs. Since your compiler doesn't generate code yet, you will just need to add logic to detect semantic errors in C- code that attempts to use these I/O routines. Declarations (prototypes) for these routines, presented in C- syntax, are shown below:

```
void output(int)
void outputb(bool)
void outputc(char)
int input()
bool inputb()
char inputc()
void outnl()
```

In order to detect errors in C- code that refers to these routines, you will need to load information about these routines into the symbol table before you parse any C- program that might use them. One way to accomplish this is to build a separate AST containing just information about these routines, then traverse this separate AST using your tree traversal routine to load the information into the symbol table. Once this is accomplished, you can then traverse the AST of the full program using the same tree traversal routine and the symbol table information loaded in the first scan will be available to be used

during semantic analysis of the full program. A graph of the AST containing the information about the I/O routines is shown below.



In the graph above, all parameters are named "*dummy*". When performing semantic analysis of C-code that uses these routines, associate line number -1 with each of the I/O routines, and feel free to continue to associate the "*dummy*" name with any formal parameters for these routines. Remember that you are not generating code for these routines, you are just detecting semantic errors/warnings in code that attempts to use these routines.

1.3 Semantic Checks

- All **while** and **if** statements must be checked to ensure that they contain predicates (or conditions) that are of type **bool**.
- All **for** statements must be checked to ensure that all expressions used in a range are of type **int**.
- The types of all operators (including assignment) must be checked to ensure that all operands are of the correct type. The types associated with expressions will have to be passed up to parent nodes (synthesized attributes). This should have been accomplished in homework 3.
- The type of any expression used in a **return** statement must be checked to ensure that it conforms with the return type of the function in which it appears.
- The number and types of all function arguments (actual parameters) must be checked to ensure that they are correct. This involves comparing the number of formal parameters with the number of actual parameters, and comparing the type of each formal parameter to the function with the type of each corresponding actual parameter to ensure that the types match. To accomplish this, you should use the symbol table to store/retrieve the TreeNode pointer associated with a given function definition.
- Functions that return a value must be checked to determine that they contain a **return** statement. It is much more difficult to determine whether all control paths in a function return a value, so for now just check to determine whether or not the function contains any **return** statement and issue a warning if none is detected. Functions that do not return a value can use an explicit **return** statement with no value provided, or can just not use a **return** statement at all.
- All **break** statements must be checked to ensure that they appear inside an iteration statement.

- All identifiers must be checked to see if the identifier has already been defined or not, and to use the symbol table to set the type of the identifier TreeNode to the type of the declaration in which the identifier appears. If the identifier is undefined, then use an "unknownType" or other marker to indicate that the type of this identifier, if not matched in a surrounding expression in a parent TreeNode, will not generate an error (this is to prevent cascading errors). Most of this should already have been done for homework 3.

- Initializers must be checked to ensure that the value of an initializer is a constant.

- Indexed identifiers (arrays) must be checked to ensure that the identifier being indexed is indeed an array. Also check to ensure that arrays without used without an index are only used when it is legal to do so. This should have already been done for homework 3.

- Check to ensure that the unary * operator (sizeof) is only used with arrays.

- Check to ensure that all "global" symbols are used in the program. If a global symbol is not referenced anywhere in a program, emit a warning of the specified format. Do not emit warnings of this type for the main function (since most programs won't call main directly) or for any of the I/O runtime routines (since they are being added regardless of whether or not the program refers to them).

2. Deliverables and Submission

Your homework will be submitted as an *uncompressed* .tar file that contains no subdirectories. Your .tar file must contain all files and code necessary to build and test your compiler. If you use ourgetopt, *be sure to include the ourgetopt files in your .tar archive***.**

The .tar file is submitted to the class submission page. You can submit as many times as you like. **The last file you submit before the deadline will be the only one graded.** No late submissions are accepted for this assignment. For all submissions you will receive email at your uidaho address showing how your file performed on the pre-grade tests. The grading program will use more extensive tests, so thoroughly test your program with inputs of your own. The error messages that your c- emits will be sorted before comparison with the expected output so that the order in which your messages are printed is not as important as it otherwise would be.

Your code must compile successfully and have no shift/reduce or reduce/reduce conflicts from bison. Your program must run with no runtime errors such as segmentation violations (SIGSEGVs).