Submission Date: 15th March 2019, Friday, 11:59:00pm

In the next milestone, you extend the project to perform semantic analysis. The goal is to convert your program into an Intermediate Representation to be used by later stages (Final code generation, optimization, etc.). Once again, you will use "actions" in your lexer/parser to achieve the desired outcome.

1. As you process the program, the information contained in the declarations is stored in the symbol table. This information includes the types of variables and functions and the sizes and offsets of variables.

2. When you process the non-declarative part, then two things happen:

(a) The information in the symbol table is used to ensure that the variables are used within the scope of their declarations and are used in a type-correct manner. If they are not, then the program is rejected.

(b) If the program is syntactically and semantically (type and scope) correct, 3-address code is generated. This 3-address

code structure (3AC) along with the information in the symbol table will be used to generate code.

Note: If you prefer some other IR over 3AC, let me know. In that case,

assume 3AC refers to your IR below.

In this assignment, we shall construct the symbol table, do the semantic checks and then generate 3AC. You will need to define the 3AC instructions corresponding to various constructs in the program.

You can create the following two-level symbol table structure.

Again feel free to adapt to nitty gritty of your language.

(a) A global symbol table (GST) that maps function names to their local symbol tables.

(b) A local symbol table for every function that contains the relevant information for the parameters and the local variables of the function.

Make sure your symbol table is "extensible" since you might discover the need to store new information as the project progresses.

Along with the construction of the symbol table, process the non-declarative part of the program to create the 3AC. The 3AC should be kept in some data structure in memory (List, Vector etc.). Perform semantic checks to ensure that semantically incorrect programs are rejected.

Your output for good programs will consist of:

(a) A dump of the symbol table of each function as a CSV file
    (the columns of CSV can be of your choice), and

(b) A dump of the 3AC of the functions in text format.

For bad programs, the output should mention the error that caused the
    program to be rejected (not just "error at line
    YYY"). Acceptable errors include (but not limited to):

"Type mismatch in line 72",
"Incompatible operator + with operand of type string",
"Undeclared variable on line 38".

Here are some more things that you have to keep in mind:

1. Operator/Function disambiguation: This will be a good time to
   pin down the exact operator function that will be used in an
   expression. For example, for x + y,: (a) If x and y are both
   ints, resolve the + to (say) +int. (b) If x and y are both
   floats, resolve the + to +float.

2. Type Casting: Continuing with the above example of x + y, if x

is an int and y is a float, cast x to a float and resolve the + to +float. The 3AC will be like:

```
t1 = cast-to-float x
t2 = t1 +float y.
```

3. Here is an (incomplete) list of errors that you have to look out for:

(a) All forms of type errors. Examples: Array index not being an integer. Variables being declared as being the void type.

(b) All form of scoping errors.

(c) Non-context-free restrictions on the language. For example, an array indexed with more indices than its dimension and functions being passed with less than the required number of parameters.