CSCI 2320: Principles of Programming Language
Programming Assignment 3: Type System and Semantic Analysis

Due date: Friday, 4/23 (11:59pm)

Points: 100

Reading: Chapters 5, 6, 7, and 8

Collaboration Policy: You are encouraged to do this assignment **in groups of two**, although individual work is also fine. If you pair up, you still need to submit your own solution based on your Assignment 2 solution. To be clear, pairing up does not mean one submission for two students. Students working in a pair can share ideas and code snippets freely between them. They can also share screen. The goal is to solve the multi-faceted problem in this assignment together. Write the name of your assignment partner on top of your source code in a comment.

Why? The ultimate reason we have studied lexical analysis and syntactic analysis is to be able to do *semantic analysis*. This is the most challenging job in programming language design. It brings the programmer and the compiler designer to the same page. It gives life (or meaning) to the code written by a programmer and thereby helps the programmer solve problems.

How is it connected to the class? This assignment is based on several recent.

How will it help my CS education? Whenever we write a line of code, we are conscious about the meaning of that line or the impact of that line on the program. This assignment gives a very different perspective on it: How does a programming language evaluate the meaning of our code? This knowledge will make us better programmers, no matter what language we use.

Submission Instruction: Submit only the .py source file on Blackboard. Use Python 3. Make sure you include the following lines so that your code (1) can be run from a terminal and (2) can take the input file as a command line argument.

```
import sys

def main(input_file_name):
  # Other code

if __name__ == "__main__":
   main(sys.argv[1])
```

Your code will be evaluated by the following command in terminal (file names are arbitrary).

```
$ python  anyfilename.py  anyinputfilename.txt
```

How to debug with a command line argument? See "Passing command line arguments to the main function" in the Visual Studio Code handout on Blackboard.

Output Specification: The last part of this document specifies the details of output.

Note on Deadline: The two-week deadline is meant for flexibility. Please aim to submit earlier.

Tasks

Build (1) a type system and (2) a semantic analyzer for the C Lite language. To do these, you should build on your syntactic analyzer (Assignment 2). Make necessary corrections based on Assignment 2 feedback.[1] As you can imagine, the input here is a stream of tokens and lexemes. Recall that lexemes were not at all used in Assignment 2. In this assignment, however, lexemes will be critically important—so important that you will need to use some specialized data structure (i.e., symbol table) to handle them.

In this assignment, some simplifying assumptions will be made regarding the source C Lite program. There will be no global variables. There will be just a single function—the main() function. Also, within the main() function, variable declarations will precede all the other statements. Therefore, even if we consider static scoping (as in C, Java, or Python), a single symbol table with just one dictionary will be sufficient for this assignment.

Part I: Type System

The job of a type system is to detect type errors. Typically, a type error means that an operator and its operand(s) are "incompatible." This brings up the question of what is compatible and what is not. Programming languages are remarkably divergent on this issue. For example, the expression `"abc"*2` will not generate any type error in Python, but it will in C. Other common type errors are using an undeclared variable, "incompatible" implicit or explicit type conversion, etc. A type system begins with a list of type rules specifying these rules of compatibility. In this assignment, you will implement the following type rules.

Type Rule 1: All variables must be declared before use
To implement this, you basically need to store some information about the variables declared in the "Declarations" part. One possibility is to build a globally accessible "symbol table" data structure. A symbol table can be implemented by a hashmap (in Java), map (in C++ STL), or dictionary (in Python). Each entry of this symbol table is a key-value pair. Here, each key is a variable name and its value is some information about the variable, such as its type and its R-value. Note that against each key (i.e., variable name), you will need to store multiple things (type and value).

Whenever you encounter an identifier in other parts of the syntax tree (e.g., in an assignment statements or in an expression), you can look up the symbol table and check whether the identifier is there. If not, you will generate a "variable not declared" error.

Symbol Table Implementation: In Python, you can use a dictionary to implement the symbol table. The key would be the variable name and the value would be the type of the variable and its

[1] Please come to office hours if you are not able to resolve all the issues with Assignment 2.

value. You can store these two in a (heterogeneous) list. The attached **`symbol_table.py`** program illustrates this.

Type Rule 2: No two variables can have the same name

You can use the symbol table to enforce this rule. Whenever a new variable is declared in the "Declarations" part, check if there already exists a variable with the same name.

Type Rule 3: Narrowing conversions are <u>not</u> allowed; widening conversions are allowed

You can do it only after implementing Part II: Semantic Analysis!

In an assignment statement, the left-hand-side variable must be big enough to hold the value generated by the right-hand-side expression. You will implement this rule only for integer and float data. Mixture of Boolean or character data with integer or float will <u>not</u> be allowed in any expression. In those cases, you will generate a type error and exit.

Example: if x is an integer variable and y is a float variable, then none of the following two statements will be allowed by the type system.

```
x = y + 2;
x = 10.0;
```

However, the following will be allowed.

```
y = x + 2;
```

To enforce this rule, each expression must also have a type, which is the "largest" type present in the expression. For example, the type of the expression `y + 2` is float, because y is a float and 2 is an integer. The main challenge in this part is to implement the type of expressions in order to enforce this rule.

Hint: The type of an "Expression" depends on the types of "Conjunctions" in it. The type of a "Conjunction" depends on the "Equalities" in it, and so on. Finally, the type of a "Factor" is the type of an identifier or a literal (such as an int, float, or bool) or the type an Expression in parenthesis.

Suggestion: You can start your implementation with a stricter version of this rule, namely, no type mismatch is allowed (e.g., for the expression x + y, both x and y have to be of same type). You can then relax it to allow widening type conversions.

Part II: Semantic Analysis

Note on terminology: **braces** are **{ }**, **parentheses** are **( )**, and **brackets** are **[ ]**.

In this part, you will keep track of the "state" of the input program from the beginning to the end. The main challenge is evaluating an expression. For example, the value of the expression (2 + 3 * 4) is 14, whereas the value of 2 + 3 * 4 > 0 is **True**. Correct evaluation is key.

For this assignment, you can expect the following types of statements—*assignment*, *if*, *print*, *return*, and *while*. The if statement will have only one statement in its body (no braces). As an example, your program should output 10 for the following CLite program (beware—the input will be tokens and lexemes as in Assignment 2, not a program like the one below). Here, the output 10 is due to the `print i;` statement. To print the value of the variable i, you will need to look up the symbol table.

```
int main()
{
      int i;
      int k;
      k = 5;
      i = 0;
      if (k > 0)
          i = 2 * k;
      print i;
      return k;
}
```

Common Mistake: Consider the Boolean expression k > 0 within the if statement above. The most common mistake is changing the value of k while evaluating such a Boolean expression. Here, k must not be changed! In fact, the state of the program must remain unchanged while evaluating any expression, be it Boolean or arithmetic! Assignment statements, of course, will change the state of the program.

Case for Thinking: Consider the following code snippet (again, beware—the input will be tokens and lexemes as in Assignment 2, not code snippet).

```
k = 50;
if (k > 100)
      i = 10;
```

Looking at the above code, it is obvious that the if condition is false, and as a result, the variable i will not be assigned to 10. Interestingly, we still need to scan the line `"i = 10;"` without executing the line (i.e., without assigning i the value of 10). How would you do this? That is, how would you parse a line like `"i = 10;"` without changing the state of the program?

Hint: The truth value of the conditional expression `"k > 100"` indicates whether we should execute `"i = 10;"` Also, regardless of this truth value, the line `"i = 10;"` must be scanned.

while loops without braces

The body of the while loop will contain just one statement (no braces). As long as the condition of the while loop is true, you will have to evaluate the body and make changes to the state of the program. For the following source program, the output of your program should be 16.

```
int main()
{
     int i;
     i = 1;
     while (i < 10)
        i = i * 2;
     print i;
     return i;
}
```

Input and Output: The input is a text file with a list of tokens and lexemes in it (tokens in the first column, lexemes second) that came from a syntactically correct CLite program. The output of your program should be type errors (if any) plus outputs generated by print statements. Outputs should be shown *on the screen*. If there is a type error, your program should exit right away.

Sample Input (input.txt file on Blackboard):

type	int
main	main
(	(
)	)
{	{
type	int
id	var1
;	;
type	float
id	var2
;	;
id	var1

```
assignOp        =
intLiteral      50
;               ;
id              var2
assignOp        =
floatLiteral    10.0
;               ;
if              if
(               (
id              var2
relOp           <
intLiteral      2
multOp          *
id              var1
)               )
id              var2
assignOp        =
(               (
intLiteral      100
addOp           +
intLiteral      2
multOp          *
id              var1
)               )
;               ;
print           print
id              var2
;               ;
return          return
id              var2
;               ;
}               }
```

Sample Output:

200.0

Design Issues:

This assignment leaves some design questions open. For example, what are compatible types for arithmetic operations? What will be the result of an integer division (such as 3/2)—will the result be float (such as 1.5) or int (such as 1)? Can a Boolean value be added to an integer resulting in an integer value? For simplicity, we will assume that you cannot add a Boolean value to an integer number, even though it's allowed in C. The main requirement here is that a float variable can be assigned integer value due to widening conversion. If you make additional design decisions, make sure that you write them down as comments at the beginning of your program.