1 Error Detection in C and WLP4

1. (a) There is no ` token in WLP4. This is a lexical error.
   (b) The variable ‘y’ is declared twice. This is a semantic error.
   (c) The variable ‘x’ is divided by 0. This is a run-time error.
   (d) The variable ‘idx’ is not declared at all. This is a semantic error.
   (e) In WLP4, all variable declarations must proceed all statements. This is a syntax error, since the
       WLP4 grammar forces this structure.
   (f) We don’t have tokens for character literals (in single quotes) in WLP4. This is a lexical error.
   (g) We have a duplicate definition of sub. Although they have different number of parameters,
       function overloading is not a feature in WLP4. This is a semantic error.
   (h) Function f is declared before function g, therefore f has no knowledge about g’s existence. Note
       that this prevents the use of mutual recursion in WLP4. This is a semantic error.
   (i) This program runs indefinitely. This is a run-time error.

2. (a) It may at first appear that there should be a semantic error when assigning the output of a
       function that returns a float to a variable of type int, however this is perfectly legal in C. In fact,
       the type of constants ‘3.0’ and ‘4.4’ are actually double, so when you return the result of ‘a * 3.0’
       or pass ‘4.4’ as a parameter to ‘triple’, there is an implicit narrowing conversion to type float! The
       error actually occurs in two places: when trying to assign the address of ‘a’ into ‘y’, and when
       trying to dereference ‘y’. This is because the type of ‘y’ is actually int, not int*, as it may first
       appear. This is a semantic error.
   (b) The function ‘getRandom()’ is not declared. This is a semantic error.

2 Symbol Table Error Checking

In the assembler, we needed to do two passes because labels could be used before they were declared. In
WLP4, we require declaration before use. That is, we know immediately that it is an error to see an unknown
identifier in any given variable use or function application. In fact, the WLP4 grammar takes care of the
former for us!
3 Type Checking

1. For an assignment statement to be well-typed, we need: - LHS and RHS are both well-typed - LHS and RHS both have the same type

Let’s consider the left-hand side first. *(d+(((c-&b)+d)-(c+(a*b))))*

We will find the types of the innermost expressions first and use them to build up the type of the full expression. We can draw this as a tree.

```
* ( d + ( ( ( c - & b ) + d ) - ( c + ( a * b ) ) ) )
  | int* - int* | | int * int
  | \ / | | \ /
  | ( int ) + int* int* + ( int )
  | \__ __/ \ /
  | ( int* ) - ( int* )
  | \___ ___/
  | \ / 
  | new int [ int ]
  | \___ ___/
  | \ / 
  | * int*
  | \ / 
  | int + int
  | \ / 
  | int
```

So the left-hand side is well-typed, and its type is int.

Now consider the right-hand side:

```
( c - d + * new int [ d + b - c ] )
int* - int* | int* + int |
\ / \ / | \ /
| int* - int* |
| \ / 
| new int [ int ]
| \ | | | * int*
| \ | | int + int
| \ / 
| int
```

The right-hand side is also well-typed, and its type is int. Therefore, the entire statement is well-typed.
2. We need to consider the types of the four expressions, then check if the types are appropriate for the statements they are contained in.

The type of \*(c+a\%b) is:
\[ \begin{array}{c}
* ( c + a \% b ) \\
| int \% int \\
| \ / \\
int* + int \\
\ / \\
* int* \\
| \\
int
\end{array} \]

The type of (&a-&b) is int* - int* = int.

Therefore we have if (int < int), which is valid because a boolean test requires both things being compared have the same type. So the if statement is well-typed.

The type of &
\[ & * & * c - ( & b ) \]

The type of *d+&a-c is:
\[ \begin{array}{c}
* d + & a - c \\
int + int* \\
\ / \\
\ / \\
int* - int* \\
\ / \\
\ / \\
int
\end{array} \]

println requires an int argument, so the println is well-typed.

3. We follow the same strategy as we did in the previous questions, but this time we added in procedure. A procedure is well-typed if the number of arguments are the same as the definition, and the type of each argument is in the same order. In WLP4, all functions are integer so we don’t have to consider the return type for the function.

We break down the expression by the arguments. The first thing we like to verify is foo(a,b), since both a and b are int, and therefore the type is correct. The expression itself is type int since all procedure in WLP4 returns type integer only.
We take a look at \texttt{foo(*c, foo(&a-&b,*\text{(c-a)}))}, the type is:

\begin{verbatim}
foo( * c , foo( & a - & b, * ( c - a )))
  int    int* - int* * (int* - int)
     |    \ /   \ /    /    /    /    / \\
     | int    * int*     \\
     \     \      /    \\
     \      int     \\
     \     /        \\
     \    /          \\
     \ foo(int, int )
     \         /      \\
     \        /        \\
     \ foo( int, int )
     \            \\
     \             int
\end{verbatim}

Since both arguments are typed \texttt{int}, thus the entire expression is well-typed.