1 Error Detection

1.1 WLP4

1. Symbol \( ^{\wedge} \) is not a valid token in WLP4.
2. The variable ‘\( y \)’ is declared twice. This is a semantic error.
3. The variable ‘\( \text{idx} \)’ is not declared at all. This is a semantic error.
4. In WLP4, all variable declarations must proceed all statements. This is a syntax error, since the WLP4 grammar forces this structure.
5. We don’t have character literals (in single quotes) in WLP4. This is a syntax error.
6. The compiler will not detect error in this program, however, this program contains an infinite loop. This is a runtime error.
7. 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.

1.2 C

1. 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 ‘\( a \ast 3.0 \)’ or pass ‘4.4’ as a parameter to ‘\( \text{triple} \)’, 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.
2. The function ‘getRandom()’ is not declared. This is a semantic error.

2 Symbol Table

1. 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 Errors

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 |

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:

\[
* ( c + a \% b ) \\
| \text{int} \% \text{int} \\
| \text{int} + \text{int} \\
\text{int} - \text{int} \\
* \text{int} \\
\text{int}
\]

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) is:

\[
& * & * c - ( & b ) \\
* \text{int} \\
/ \text{int} \\
/ \text{int} \\
* \text{int} \\
/ \\
& \text{int} \\
/ \\
\text{int}
\]

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

The type of *d+&a-c is:

\[
* d + & a - c \\
\text{int} + \text{int*} \\
\text{int*} - \text{int*} \\
\text{int}
\]

However, the delete [] operator is for deleting arrays, so it requires an int* type. Thus the statement delete[]*d+&a-c is not well-typed.