1 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!

2 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* )
  |       \ \ -/- \ / 
  |     \ / 
  | int* + ( int )
  \----- ---/-
      \----- ---/-
         \ / 
* ( int* )
  | int
```

So the left hand side is well-typed, and its type is int.
Now consider the right hand side:

```plaintext
(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:

```plaintext
* ( c + a % b )
 | int % int
 | \ /    int* + int
|   \ /    *
| * int*
|   |
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:

```plaintext
& * & * c - ( & b )
 * int*   |
   |   |
& int    |
   |   |
   |   *
int*    |
   /    |
   /    |
& int  |
   /    |
   int* - int*
   \--   --/ 
    \ / 
    int
```

println requires an int argument, so the println is well-typed.
The type of \(*d+&a-c\) is:

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

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

3 Pre- and Post-Increment Code Generation

3.1 Pre-Increment Code Generation

First recall what these two operators do. \(+ + i\) increases the value of \(i\) by 1 and then returns the new value, while post-increment increases the value of \(i\) by 1 but returns the old value.

Pre-increment is a bit more straightforward:

```c
void genCode(tree t)
{
    if (t.rule is "factor -> PLUS PLUS lvalue"){
        // generate code to put the lvalue in $3
        // You'll do this in your assignment
        genCode(t.children[2])
        // $3 contains an lvalue, which means we need to fetch the actual value
        // from memory
        // We're making the assumption that $3 is the full address and not just an offset
        // Otherwise we need to do something like: add/sub $3, $3, $29
        lw $5, 0($3)
        // Add one to the value
        add $5, $5, $11
        // Save the new value
        sw $5, 0($3)
        // Return new value
        add $3, $5, $0
    }
}
```

Note that this is a factor and not an lvalue, so we don't need to return a memory address. This also means we can't nest increments (for example, \;+++i\).
3.2 Post-Increment Code Generation

Post-increment requires that we remember the old value:

```c
void genCode(tree t)
{
    ... 
    if(t.rule is "factor -> lvalue PLUS PLUS"){
        // generate code to put the lvalue in $3
        genCode(t.children[0])
        // $3 contains an lvalue, which means we need to fetch the actual value
        // from memory
        // We're making the assumption that the full address and not just an offset
        // Otherwise we need to do something like: add/sub $3, $3, $29
        lw $5, 0($3)
        // Copy the old value
        add $6, $5, $0
        // Add one to the value
        add $5, $5, $11
        // Save the new value
        sw $5, 0($3)
        // Return old value
        add $3, $6, $0
    }
}
```

As you can see, post-increment is not much more expensive than pre-increment for integers. It might be considerably more expensive for more complex data structures, though. As an exercise, do post-increment without saving a copy of the old value.