ECE 468 & 573
Problem Set 3: Symbol tables, ASTs and semantic actions

1. Why do we track the number of dimensions and sizes of arrays in symbol tables?

   Answer: Two reasons. First, we need to know how big arrays are so that we can correctly allocate space for them in activation records. Second, we need to know the dimensions and sizes of multi-dimensional arrays so that we can correctly generate code that accesses them. Consider the code `A[i][j]`. To access this element of the array, the compiler generates the following address calculation code: `&A + i * dim1 + j`. This requires knowing dim1.

2. What differentiates an abstract syntax tree from a parse tree?

   Answer: A parse tree captures the structure of the program according to its grammar, with interior nodes for non-terminals, and leaves for terminals. The same language, parsed with a different grammar, would produce a different parse tree. An abstract syntax tree, on the other hand, captures the structure of the program independent of the grammar. This structure is often more useful: expressions can be represented as expression trees, with interior nodes being operations, for example.

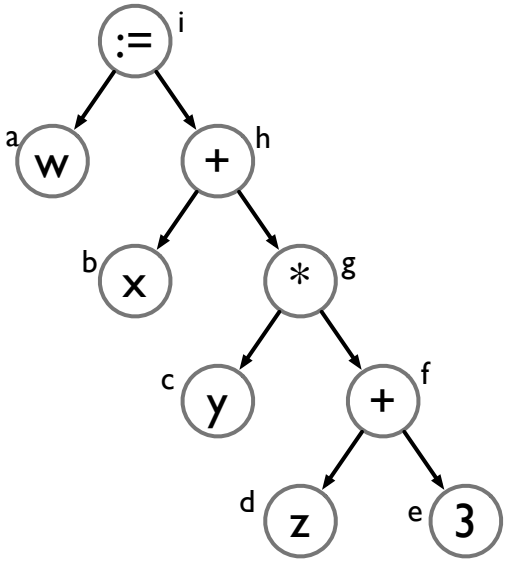
3. Name one advantage to generating ASTs before producing code, rather than producing code directly.

   Answer: There are several advantages. Many compilers make passes over the AST to check for correctness (e.g., to make sure that everything is well typed). Many optimizations take place at the AST level, when the structure of the code is still apparent.

4. Show what the abstract syntax tree would look like for the following expression:

$$w := x + y * (z + 3)$$

   Answer: The letters beside the nodes are the names of the nodes, which will be used in later answers.



5. Give three address code would be generated for the above tree. Use the following instructions: **LD A, T** loads from variable A into temporary T. **OP T1, T2, T3** performs `T3 = T1 OP T2`. **ST T, A** stores from variable A into temporary T.

   Answer:

   If you generate the code manually, you would probably come up with something like this:

   ```
   LD X, T1
   LD Y, T2
   LD Z, T3
   ADD T3, 3, T4
   MUL T2, T4, T5
   ADD T1, T5, T6
   ST T6, W
   ```

   If you generate the code automatically, you will probably come up with something like this:

   ```
   LD x, T6
   LD y, T4
   LD z, T2
   ADD T2, 3, T1
   MUL T4, T1, T3
   ```

```
ADD T6, T4, T5
ST T5, w
```

See the answer to the next problem to see how this might happen.

6. Show the code generation information (any code, what temporary stores the result, and whether it's an l-value or an r-value) for each node in the AST above.

   Answer: Note that in these code generation examples, I am (a) deferring loads until we know whether a variable is used as a source or a destination for an assignment, (b) generating the temporary for the result of an expression before generating any code for the expression, including any loads that might need to happen. The AST nodes are traversed in post-order, visiting the left child before the right child (i.e., in alphabetical order)

   Node a:

   ```
   Temp: w
   Type: l-value
   Code: ---
   ```

   Node b:

   ```
   Temp: x
   Type: l-value
   Code: ---
   ```

   Node c:

   ```
   Temp: y
   Type: l-value
   Code: ---
   ```

   Node d:

   ```
   Temp: z
   Type: l-value
   Code: ---
   ```

   Node e:

   ```
   Temp: 3
   Type: constant
   Code: ---
   ```

Node f:

```
Temp: T1
Type: r-value
Code: LD z, T2
      ADD T2, 3, T1
```

Node g:

```
Temp: T3
Type: r-value
Code: LD y, T4
      LD z, T2
      ADD T2, 3, T1
      MUL T4, T1, T3
```

Node h:

```
Temp: T5
Type: r-value
Code: LD x, T6
      LD y, T4
      LD z, T2
      ADD T2, 3, T1
      MUL T4, T1, T3
      ADD T6, T4, T5
```

Node i:

```
Temp: N/A
Type: N/A
Code: LD x, T6
      LD y, T4
      LD z, T2
      ADD T2, 3, T1
      MUL T4, T1, T3
      ADD T6, T4, T5
      ST T5, w
```

Note that node i represents a complete statement, and hence does not have a temporary holding its value.