**Introduction**

The purpose of my project is to create a new language, let's give it the eponymous name of AP++, that consists of a subset of features and syntactical sugar of the various languages for which I have developed a predilection; from Python's intuitive list slicing syntax to the ++ operator and {} scope blocks in C++. AP++ will be much smaller in scope than the gamut of features offered in these modern programming languages, but will still be significant enough to be able to implement a number of algorithms.

**Language Features**

*Scope*

{} blocks for defining scope
; line termination

*Comments*

// single-line comments

*Conditionals*

if (conditional expression1) {
} else if (conditional expression2) {
} else {
}

*Variables*

Variables will be strictly typed in AP++,
e.g. `int x = 4; bool y = true; void foo(int x);`

2 basic primitive types: Integer, Boolean

*Integer (keyword int)*

Declaration:
e.g. `int x = 1; int x = y; int x = y + 1;`

*Operators:*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Arithmetic Addition</td>
<td><code>x + y : between 2 vars</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x + 1 : between var and literal</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1 + 2 : between 2 literals</code></td>
</tr>
<tr>
<td>-</td>
<td>Arithmetic Subtraction</td>
<td><code>x - y : between 2 vars</code></td>
</tr>
</tbody>
</table>
### Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Arithmetic Division</td>
<td>x / y: between 2 vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x / 1: between var and literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 / 2: between 2 literals</td>
</tr>
<tr>
<td>*</td>
<td>Arithmetic Multiplication</td>
<td>x * y: between 2 vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x * 1: between var and literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 * 2: between 2 literals</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>x % y: between 2 vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x % 2: between var and literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 % 2: between 2 literals</td>
</tr>
<tr>
<td>++x</td>
<td>Unary Pre-Increment Operator</td>
<td>++x</td>
</tr>
<tr>
<td>x++</td>
<td>Unary Post-Increment Operator</td>
<td>x++</td>
</tr>
</tbody>
</table>

### Boolean

(keyword: `bool`, values: `{true, false}`)

Declaration:
e.g. `bool x = true; bool x = y; bool x = conditional expression;`

### Operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Boolean AND</td>
<td>x &amp;&amp; y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Boolean NOT</td>
<td>!x</td>
</tr>
</tbody>
</table>

Variables declared outside of a scoped block `{}` will be considered global variables that live on the heap. All other variables will be allocated on the stack.

### Lists

Python-style mutable lists.

Declaration:
e.g. `int x[] = []; int x[] = [1, 2, 4]; int x[] = y[:];`
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list.append(x)</td>
<td>appends element x to end of list</td>
</tr>
<tr>
<td>list.insert(i, x)</td>
<td>inserts element x at ith index</td>
</tr>
<tr>
<td>list.pop([i])</td>
<td>pops ith element of list of i specified, else from end</td>
</tr>
<tr>
<td>list.clear()</td>
<td>clears all elements from list</td>
</tr>
<tr>
<td>[:] splicing</td>
<td>returns sublists of specified range, e.g.</td>
</tr>
<tr>
<td></td>
<td>l[: ] - returns new list with all elements from l</td>
</tr>
<tr>
<td></td>
<td>l[4:] - returns elements from index 4 to last</td>
</tr>
<tr>
<td></td>
<td>l[:4] - returns elements from index 0 to 4 index</td>
</tr>
<tr>
<td></td>
<td>inclusive</td>
</tr>
<tr>
<td></td>
<td>l[2:4] - returns elements from index 2 to 4</td>
</tr>
<tr>
<td></td>
<td>inclusive</td>
</tr>
</tbody>
</table>

**Loops**

```python
while (conditional expression) {
}
```

I opted not to implement the for loop since the same functionality can be achieved with a while loop and local variables.

**Functions**

```python
with return types
int foo(int a, int b) {
    return 0;
}
```

no return types:
```python
void foo() {
}
```

There will be no support for default arguments, variable arguments or function overloading. Every param and return will pass by value, not reference.
Example Programs

**Euclidean Algorithm (GCD)**

```c
int gcd(int x, int y) {
    if (y == 0) {
        return x;
    }
    return gcd(y, x % y);
}
```

**Merge Sort**

// merges two sorted sublists of arr[] (arr[0..m], arr[m+1..r]) in-place.
void merge(int[] arr, int l, int m, int r) {
    // temp lists for l and r sides
    int[] L = arr[0:m];
    int[] R = arr[m+1:r];

    // merge the temp lists back into arr[l..r]
    int i = 0;     // init index of 1st sublist
    int j = 0;     // init index of 2nd sublist
    int k = l;     // init index of merged sublist

    while (k < r) {
        if (j >= r || (i < m && L[i] <= R[j])) {
            arr[k] = L[i];
            i++;
        } else if (i >= m || (j < r && L[i] > R[j])) {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
}

void mergeSort(int[] list, l, r) {
    if (l >= r) {
        return;
    }
    int m = (l + (r-1)) / 2;
    mergeSort(list, l, m);
    mergeSort(list, m+1, r);
    merge(list, l, m, r);
}