Basic syntax for C0 programs

Semicolons: statements are terminated by semicolons. At the end of most lines, you'll need a semicolon. (Exceptions are if statements, function definitions, use statements, and loops.)

Types: Some of the types in C0 are:

- **int**: Integers $x$, where $-2^{31} \leq x < 2^{31}$.
- **bool**: Either true or false. Useful for conditionals, loops, and more.
  - $a \text{ || } b$ is true if either $a$ or $b$ are true
  - $a \&\& b$ is true if either $a$ and $b$ are true
  - $!a$ is true if $a$ is false
  - $x < y$, $x <= y$, $x > y$, $x >= y$ all compare two integers $x$ and $y$ and return a bool
  - $x == y$ and $x != y$ compare most C0 types and return a bool. You can't compare strings this way, though, and it's usually bad style if you're writing code like $a == \text{ true}$.
- **string**: An ordered sequence of characters like “Hello!”
- **char**: A single character, like 'c', 'z', 'F', or '?'.

Boolean operators: Both && and || are short-circuiting infix operators.

They are infix (like other operators +, -, %, etc.) because they take two arguments and the operator is placed between the two arguments. The compiler mentions the word "infix operator" if you make a mistake with them, so it's good to be aware of this name for them.

They are short-circuiting because, if the expression to the left of && evaluates to false or if the expression to the left of || evaluates to true, then the expression to the right will never get executed. This means that, even though evaluating the expression $y/x == 0$ will cause an error if $x$ is zero, evaluating the expression $x == 0 || y/x == 0$ can never cause an error.

Locals: locals (confusingly also called “local variables,” “assignable variables,” “assignables,” or “variables”) are explicitly declared along with their type. Locals can never change type after they are declared.

```
1 int x = 5; // x is initialized to 5
2 int y;  // We can't use y until we assign to it!
3 string str = "hello";
4 y = x + 4; // y is now equal to 9
5 x = x + 1; // x is now equal to 6
6 x = "world"; // ERROR! string and int are different types!
```

Conditionals: These are one way we use bool values. Here's an example of if statements in C0:

```
1 if (condition) {
2     // do something if condition == true
3 }
4 else if (condition2) {
5     // do something if condition2 == true and condition == false
```
Loops: There are two kinds of loops in C0—while loops and for loops.

- **while loop**: It takes a condition (something that evaluates to a Boolean). The loop executes until the condition is false.

- **for loop**: It takes three statements separated by semicolons. Execute the first statement once at the beginning of the loop, loop until the second statement (a condition) is false, and execute the third statement at the end of each iteration.

<table>
<thead>
<tr>
<th>while loop</th>
<th>for loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int x = 0;</td>
<td>1 for (int x = 0; x &lt; 5; x++) {</td>
</tr>
<tr>
<td>2 while (x &lt; 5) {</td>
<td>2 printf(x);</td>
</tr>
<tr>
<td>3 printf(x);</td>
<td>3 printf(&quot;\n&quot;);</td>
</tr>
<tr>
<td>4 printf(&quot;\n&quot;);</td>
<td>4 }</td>
</tr>
<tr>
<td>5 x++;</td>
<td></td>
</tr>
<tr>
<td>6 }</td>
<td></td>
</tr>
</tbody>
</table>

These two examples do the same thing. Here, the for loop is preferred but there are cases (like binary search in an array, which we'll discuss later this semester) where while loops are cleaner.

**Function definition**: This example defines a function called add that takes two ints as arguments and returns an int.

```c
1 int add (int x, int y) {
2     return x + y;
3 }
```

**Comments**: use // to start a single line comment and /* ...*/ for multi-line comments. It's good style to have a * at the beginning of each line in a multi-line comment.

**Indentation and braces**: Your code will still work if it's not indented well, but it's really bad style to indent poorly. Python's indentation rules are good and you should generally follow them in C0 too. C0 uses curly braces (i.e. { } and }) to denote the starts and ends of blocks, as seen above. For single-line blocks it's possible to omit the curly braces, but that can make debugging very difficult if you later add in another line to the block of code. For that reason, we **highly** encourage you to always use braces, even for single-line statements.
Checkpoint 0

Identify and correct the syntax errors in the following code to make it valid C0:

```c
1 #use <conio>
2
3 def fib(i):
4     if (i == 0 or i == 1):
5         return i;
6     return fib(i−1) + fib(i−2)
7
8 int main():
9     for int i =0; i < 10; i ++
10         print (fib(i))
11         print('
')
12 return 0
```

Contracts

There are 4 types of annotations in C0 (for convenience, we’re using exp here to mean any Boolean expression):

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>//@requires exp;</td>
<td>before function execution</td>
</tr>
<tr>
<td>//@ensures exp;</td>
<td>before function returns</td>
</tr>
<tr>
<td>//@loop_invariant exp;</td>
<td>before the loop condition is checked</td>
</tr>
<tr>
<td>//@assert exp;</td>
<td>wherever you put it in the code</td>
</tr>
</tbody>
</table>

There are certain special variables and functions you have access to only in annotations. One of these is \result. It can be used only in //@ensures statements and it will give you the return value of the function. (There are other such variables/functions that we’ll get to later in the semester.)

To help you develop an intuition about contracts, here are some explanations of the different kinds of annotations:

- //@requires: For checking __________________
- //@ensures: For checking __________________
  Allow use of the special expression __________________
- //@loop_invariant: We can only write these immediately after the beginning of a while loop or for loop.
  When are these checked? ____________________________
- //@assert: Assertion statements don’t play the special role in reasoning that //@requires, //@ensures, and //@loop_invariant statements do. They can be very helpful for debugging code and summarizing what you know, especially after a loop.

Checkpoint 1

What command would you compile with to enable contract checking in a file named fastpow.c0?
**Proving correctness of the mystery function**

We use contracts to both test our code and to logically reason about code. With contracts, careful reasoning and good testing both help us to be confident that our code is correct.

Here’s a different way of looking at the mystery function from lecture yesterday. Once we have loop invariants for the mystery function, we can view the whole thing as a control flow diagram:

The circle labeled 1 is a ____________ of the function, and the circles labeled 2 and 3 are ____________. The circles labeled 4 and 5 just capture information we get from the result of the loop guard (or loop condition), but we might write 4 as an ______________ statement.

To prove this function correct, we need to reason about the two pieces of code (pieces that this diagram hides in the two cloud-bubbles) to ensure that our contracts never fail:

- When we reason about the upper code bubble, we assume that ______ is true before the code runs and show that ______ and ______ are true afterwards.
- When we reason about the lower code bubble, we assume ______, ______, and ______ are true before the code runs and show that ______ and ______ are true afterwards.
- To reason that the returned value \( r \) is equal to \( b^y \), we combine the information from circles ______ and ______ to conclude that \( y = 0 \). Together with the information in circle _______, this implies that \( r = b^0 \).

In addition, we have to reason about termination: every time the lower code bubble runs, the value \( y \) gets strictly smaller, because \( y/2 < y \) for every \( y > 0 \), and the loop invariant ensures \( y \) never becomes negative.
To summarize, in general there are four steps for proving the correctness of a function with one loop using loop invariants:

- Preservation of loop invariants

Preserving loop invariants can be a bit confusing, because we have to assume that the loop invariant, like $y \geq 0$ or $r \ast \text{POW}(x,y) = \text{POW}(b,e)$ is true before the loop invariant is checked by assuming that the loop invariant was true the last time the loop invariant was checked (we also assume that the loop guard subsequently evaluated to true).

```
1 while (y > 0)
2   //@loop_inv y >= 0;
3   //@loop_inv r * POW(x,y) == POW(b,e);
4   {
5     r = r \ast x;
6     y = y - 1;
7     x = x;
8   }
```

When an arbitrary loop begins, we know and .

After an arbitrary iteration of the loop, we use primed values to represent the new values in terms of the old ones:

$x' =$

$y' =$

$r' =$

We need to show that .

This is true because .

This terminates because .

Because we haven't changed any loop invariants, the rest of the correctness proof for exponentiation is the same as it was in class. By keeping the loop invariant the same, we still have a proven-correct function, even though we tore out loop body and replaced it with a different (and less efficient) one!