15-122: Principles of Imperative Computation — Spring 2020

Recitation 1: C0 Basics — Thursday January 16th

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 || b` is true if either `a` or `b` are true
  - `a && b` is true if both `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 `e == true`.
- **`string`**: An ordered sequence of characters encloded in double quotes like `"Hello!"`
- **`char`**: A single character enclosed in single quotes, 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 (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:

```
if (condition) {
    //do something if condition == true
}
else if (condition2) {
    //do something if condition2 == true and condition == false
}
else {
    //do something if condition == false and condition2 == 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.

while loop

```
int x = 0;
while (x < 5) {
    printint(x);
    print("\n");
    x++;
}
```

for loop

```
for (int x = 0; x < 5; x++) {
    printint(x);
    print("\n");
}
```

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 **int**s as arguments and returns an **int**.

```
int add (int x, int y) {
    return x + y;
}
```

Comments: Use `//` to start a single line comment and `/* ... */` for multi-line comments.

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, you may want to use braces, even for single-line statements.

Very Bad

```
if (x == 4)
println("x is 4");
```

Okay

```
if (x == 4)
    println("x is 4");
```

Good

```
if(x == 4) {
    println("x is 4");
}
```

Checkpoint 0

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

```
#use <conio>

def fib(i):
    if(i == 0 or i == 1){
      return i;
    }
return fib(i - 1) + fib(i - 2)

int main():
  for int i=0; i < 10; i++
  printint(fib(i))
  print(\n)
  return 0
```

Contracts

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

Annotation	Checked
`//@requires exp;`	before function execution
`//@ensures exp;`	before function returns
`//@loop_invariant exp;`	before the loop condition is checked
`//@assert exp;`	wherever you put it in the code

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

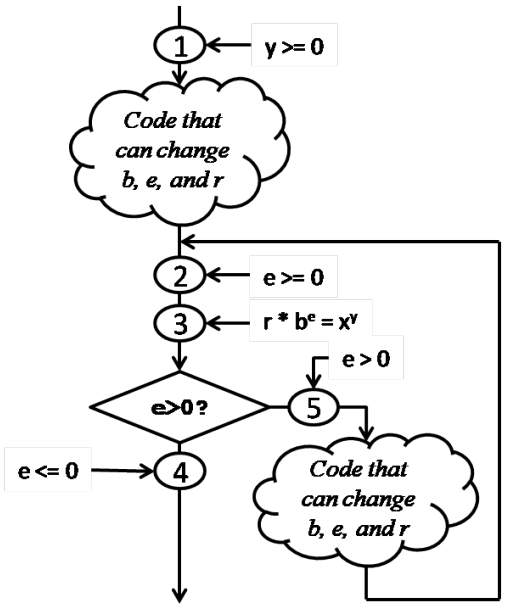
Recall lab 1. What command would you compile with to enable contract checking in a file named `fastpow.c0`?

```
prompt> _____
```

Proving the correctness of the mystery function

We use contracts to both test our code and to reason about code. With contracts, careful point-to 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. 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. (This is INITialization.)
- When we reason about the lower code bubble, we assume _____, _____, and _____ are true before the code runs and show that _____ and _____ are true afterwards. (This is PREServation.)
- To reason that the returned value r is equal to x^y, we combine the information from circles _____ and _____ to conclude that $e = 0$. Together with the information in circle _____, this implies that $r = x^y$. (This is correctness on EXIT.)

In addition, we have to reason about TERMination: every time the lower code bubble runs, the value e gets strictly smaller, because $e/2 < e$ for every $e > 0$, and the loop invariant ensures e 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

Showing that a loop invariant is preserved can be a bit confusing: we need to assume that the loop invariant, like `e >= 0` or `r * POW(b,e) == POW(x,y)` is true just before we start executing the body of the loop (it is checked just before the loop guard), and use this information (together with the knowledge that the loop guard evaluated to `true`) to show that it is also true at the end of the loop (just before the loop guard is evaluated again). We do this relative to an *arbitrary* iteration of the loop. Here's a *different* loop body from what we saw in class.

```
1    while (e > 0)
2    //@loop_invariant e >= 0;
3    //@loop_invariant r * POW(b, e) == POW(x, y);
4    {
5        r = r * b;
6        e = e - 1;
7        b = b;
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:

$b' =$ _____

$e' =$ _____

$r' =$ _____

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!