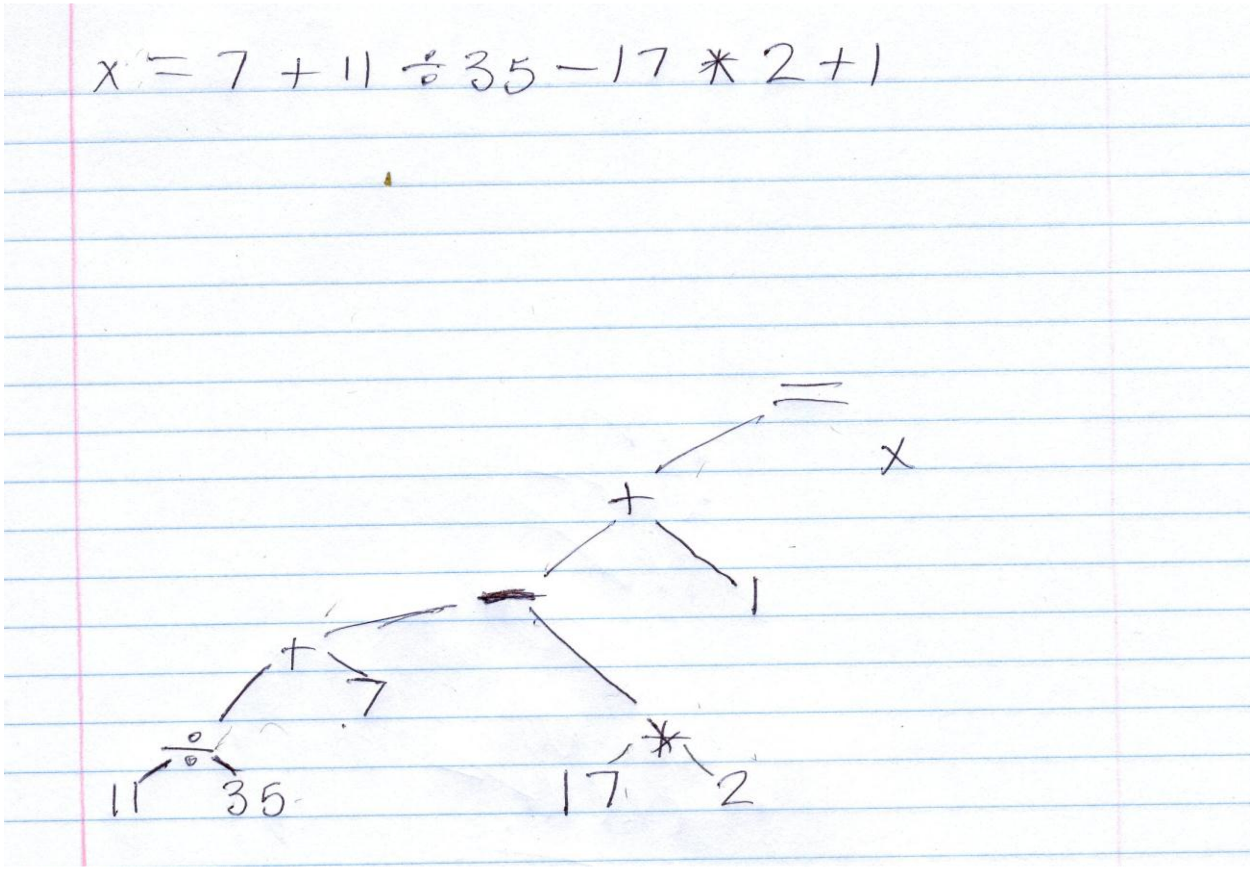
Name: Benjamin Adamson

Date: Sunday, January 22, 2012

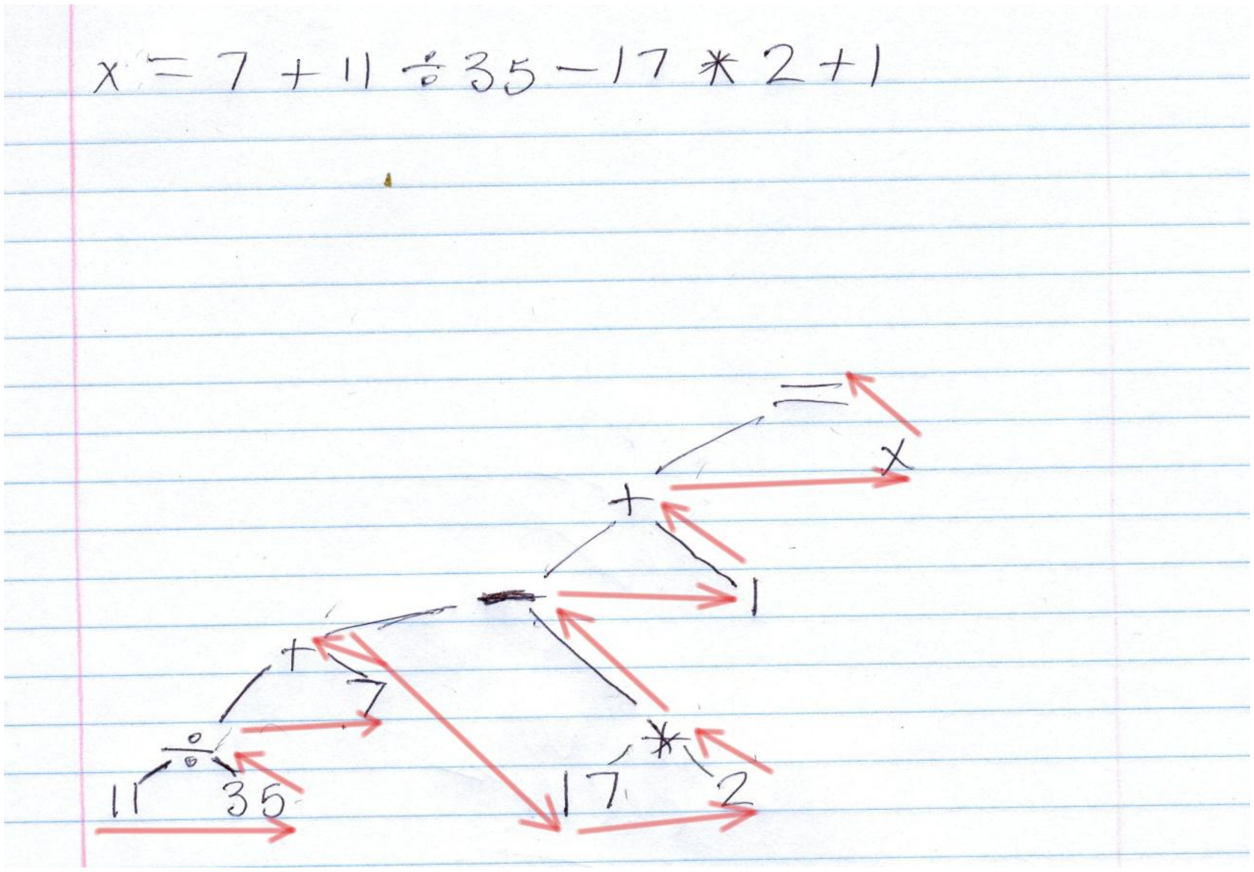
Milestone 1 Report

Handwritten Answers to Milestone Questions:

3. Translate an infix style expression to an expression tree. The output here is a drawn tree.



4. Do a post order traversal of the expression tree to generate the *gforth* input. The output of this step is *gforth* code.



Gforth output code:

```
: x 11e 35e f/ 7e f+ 17e 2e f* f- 1e f+ ;
```

Specification (what do you think the purpose of this milestone is)

I believe the milestone is dual-purposed. One, to get us used to the way assignments will work in this class. Second, I believe the milestone is assigned to get us familiar with the gforth language, and how to think in the context of the language.

Processing (how did you and/or your team go about solving the problem)

I spent the first weekend after the assignment was posted reading the documentation, with gforth. Eventually able to follow the examples in the documentation, I worked my way through the 12 examples. Now that I am done with the 12 example questions, I believe I have a firm understanding on the ‘basics’ of gforth.

Testing Requirement (how did you go about testing for correctness)

I wrote a bash script that invokes my solutions with various inputs, against known outputs. The makefile invokes my bash script, which creates the actual file stoutest.out. The first nine solutions were simple, to verify, checking the output matched the expected output was enough. For the last three (10, 11, and 12) I tested inputs that I knew against an oracle (the correct solutions in this case).

Retrospective (what did you learn in this milestone)

In this milestone I learned that just because a language seems to suck at first, doesn't mean it actually does. Once you spend some time to learn and understand how it is actually working, you can be surprised. When I accidentally redefined the word '5' to an expression, I understood how cool this language actually was. Aside from learning how gforth worked, I learned how to think in a stack-based context. Thinking about how to solve the Fibonacci series with a stack was mind-bending at the beginning to stay the least. I also had to learn / evaluate how much testing was appropriate. The tree-traversal was review from theory of computation.

Important note: The following data structure for an n-ary tree (and the algorithm that initializes/traverses it) are contained within the tar file submitted. The makefile automatically makes an executable, n-ary-tree.out that can be run. To change the depth of the tree, or n (number of nodes, or the n in n-ary) simply change the corresponding #define at the top of n-ary-tree.c

data structure for an *n*-ary tree:

```
struct node {
    struct node *parent;
    struct node *children;
    int visited;
} node;
```

pseudo-code recursive algorithm to translate an arbitrary instance of these trees into postorder

```c
// Author: Benjamin Adamson
// Class: CS480 - Milestone 1
#include <stdio.h>
#include <stdlib.h>
#define n 4
#define max_depth 2
static int global_value = 1;
void visit_child_nodes(struct node *p)
{
    int i = 0; // loop iterator

    // base case, is leaf-node
    if(p->children == NULL) {
        p->visited = 1;
        fprintf(stdout, "%d\n", p->visited);
        return;
    }

    // non-leaf
    for(i = 0; i < n; ++i)
    {
        visit_child_nodes(p->children + i);
    }

    // visit the non-leaf node (post-order)
    p->visited = 1;

void init_tree(struct node *p, unsigned int depth)
{
    int i = 0; // loop iterator

    p->visited = 0;

    if(depth == max_depth) {
        p->children = NULL;
        return;
    }

    p->children = (struct node*)malloc(sizeof(struct node) * n);
    for(i = 0; i < n; ++i) {
        init_tree(&p->children[i], depth+1);
    }
}

int main(int argc, char *argv[])
{
    struct node root;

    init_tree(&root, 0);
    visit_child_nodes(&root);

    return 0;
}
```