CPSC-354 Report

Darren Pak
Chapman University

October 3, 2022

Abstract

This is a culmination of all assignments and reports for CPSC-354 taught by Alex Kurz at Chapman University Fall 2022.

Contents

1 Introduction

My name is Darren Pak and I am a computer science major at Chapman University with a minor in Data Analytics. My current goals as of Fall 2022 are to find interesting job opportunities and career paths that I find enjoyable and are able to sustain my lifestyle.

2 Homework

This section contains solutions to homework assignments.

2.1 Week 1

In Week 1, I will go over Euclid's Algorithm for Greatest Common Divisor and how it is implemented in C++.

2.1.1 Euclid's Algorithm

Euclid's Algorithm is defined as follows:

gcd(a,b):

Input: Two whole numbers (integers) called a and b, both greater than 0.

(1) if

$$a > b$$

then replace a by a-b and go to (1).

(2) if

$$b > a$$

then replace b by b-a and go to (1).

Output: a

As described in Alex Kurz Homework (Week 1)

2.1.2 Implementation in C++

Below is the example for implementation of Euclid's Algorithm in C++:

```
#include <iostream>

using namespace std;

int gcd(int a, int b) {
    while ((a != 1) && (b != 1)) {
        if (a > b) {
            a = a-b;
        }
        if (b > a) {
            b = b-a;
        }
        if (a == b) {
            return a;
        }
    }
    return a;
}

int main()
{
    cout<<gcd(9,33)<<endl;

    return 0;
}
```

In the function gcd, there is a while loop checking if either of the inputs are 1. This will eliminate cases where the GCD is already the lowest possible and as described in Euclid's algorithm, will return 1. Within the while loop, we go over the two rules in Euclid's Algorithm. The first being if a is greater than b then a is assigned to a - b. The second rule being if b is greater than a then b is assigned to b - a. Next we resolve the output and return a as the result.

2.2 Week 2

Week 2 is focused on recursion and functions in Haskell. For this assignment, I created 6 different functions using recursion in Haskell. Find the full Github Repository here.

2.2.1 Function Select Evens

Here is a code snippet from the above mentioned Github Repository of the Select Evens function.

```
select_evens :: [a] -> [a]
select_evens [] = []
select_evens (x:xs)
    |mod (length xs) 2 == 1 = x : select_evens xs
    |otherwise = select_evens xs
```

This function takes a list as an input and returns a list of only the even index elements. For example, from a list of ["a","b","c","d"] the function would return ["b","d"]. The first line of this function determines the input and outputs which are both lists of elements. The second line determines that an empty list from the function returns an empty list. This will become our indicator for ending recursion. Next we have an if statement saying that after the head, if there are an odd number of elements remaining, then the head element is of an even index. This means it would be appended to the returning list. If there is an even number of elements remaining, this means that the head element is of an odd index, meaning that the head element is skipped and will not be appended to the list. After all of the calculations have completed, the elements are appended to an empty list and added to the front in the order they were calculated.

Collaborated with Adrian Edralin for Week 2 Assignment.

2.3 Week 3

Week 3 assignment is focused around the Towers of Hanoi solving algorithm and how to evaluate functions. This game functions with n number of rings and 3 poles where the objective is to move all of the rings from the first pole (0) to the last pole (2). However, you are not able to stack larger rings on top of smaller rings while only moving 1 ring at a time. This game can be played at mathisfun.com in a simulated environment with different numbers rings.

2.3.1 Rules of the Algorithm

Our algorithm follows the following rules to solve this puzzle as described in Towers of Hanoi (Week 3):

```
hanoi 1 x y = move x y

hanoi (n+1) x y =
   hanoi n x (other x y)
   move x y
   hanoi n (other x y) y
```

When expanded for n = 5 (5 rings), this becomes:

```
hanoi 5 0 2
  hanoi 4 0 1
    hanoi 3 0 2
      hanoi 2 0 1
        hanoi 1 0 2 = move 0 2
        move  0 1
        hanoi 1 2 1 = move 2 1
      move 0 2
      hanoi 2 1 2
        hanoi 1 1 0 = move 1 0
        move  1 2
        hanoi 1 0 2 = move 0 2
    move 0 1
    hanoi 3 2 1
        hanoi 2 2 0
            hanoi 1 2 1 = move 2 1
            move 2 0
            hanoi 1 1 0 = move 1 0
        move 2 1
        hanoi 2 0 1
            hanoi 1 0 2 = move 0 2
            move 0 1
            hanoi 1 2 1 = move 2 1
  move 0 2
  hanoi 4 1 2
      hanoi 3 1 0
        hanoi 2 1 2
            hanoi 1 1 0 = move 1 0
            move 1 2
            hanoi 1 0 2 = move 0 2
        move 1 0
        hanoi 2 2 0
            hanoi 1 2 1 = move 2 1
            move 2 0
            hanoi 1 1 0 = move 1 0
    move 1 2
    hanoi 3 0 2
        hanoi 2 0 1
            hanoi 1 0 2 = move 0 2
            move 0 1
            hanoi 1 2 1 = move 2 1
        move 0 2
        hanoi 2 1 2
            hanoi 1 1 0 = move 1 0
            move 1 2
            hanoi 1 0 2 = move 0 2
```

This eventually gets simplified to the following moves where

$$x- > y$$

defines x being a ring moving from tower x to tower y:

```
 0->2
0->1
2->1
```

```
0->2
1->0
1->2
0->2
0->1
2->1
2->0
1->0
2->1
0->2
0->1
2->1
0->2
1->0
1->2
0->2
1->0
2->1
2->0
1->0
1->2
0->2
0->1
2->1
0->2
1->0
1->2
0->2
```

2.3.2 Analysis Questions

In our original algorithm, it is shown that "hanoi" shows up 31 times. This is as the number of moves it takes to solve the puzzle meaning that the "hanoi" shows up the same number of times as the number of moves.
For 1 ring, this is simple and would only take 1 move to solve. For 2 rings, this is 3 moves to solve. For 3 rings it is 7 moves to solve, for 4 rings it is 15 moves, and for 5 rings it is 31 moves. In this we see a pattern where for each additional ring, you double the amount of moves and add 1. In conclusion if n is the number of rings, this leads us to the equation of moves(n) = 2 * moves(n - 1) + 1, where moves(1) = 1 and n is greater than 0.

2.4 Week 4

This week is focused on concrete and abstract syntax trees which are essentially ways to parse equations.

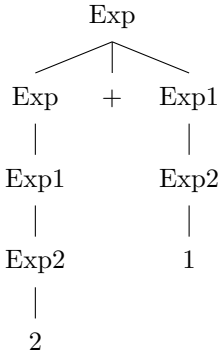
2.4.1 Concrete Syntax Trees

For the concrete syntax trees, we follow the rules and instructions as listed in the Week 4 Report Instructions. The rules are listed as following:

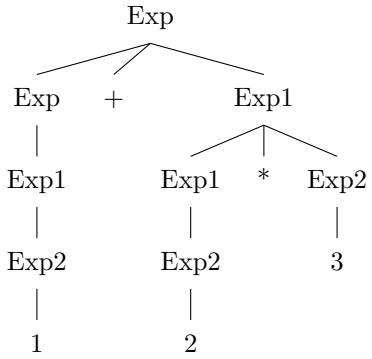
```
Exp -> Exp '+' Exp1
Exp1 -> Exp1 '*' Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'
Exp -> Exp1
```

`Exp1 -> Exp2`

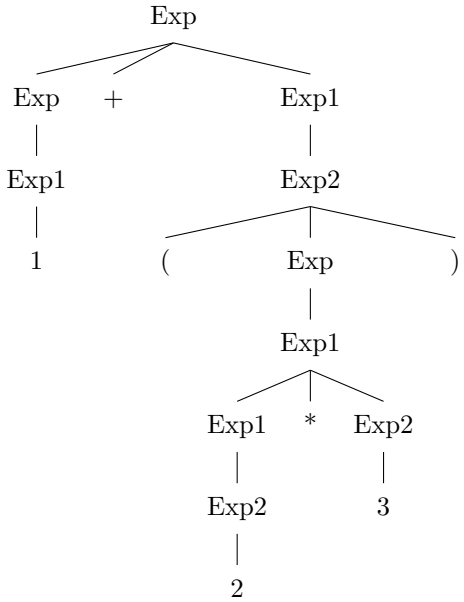
2+1:



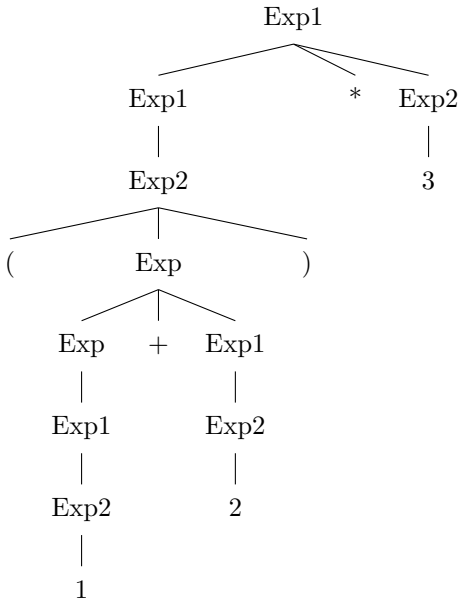
1+2*3:



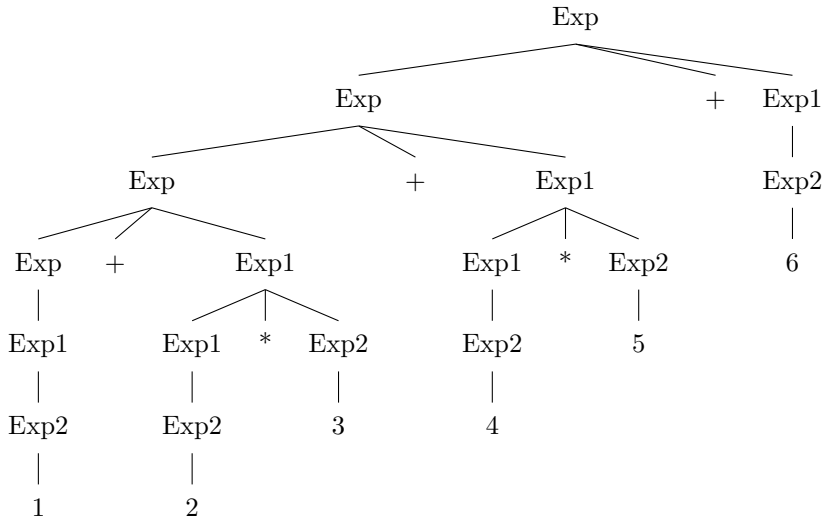
1+(2*3):



(1+2)*3:



1+2*3+4*5+6:



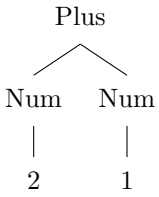
2.4.2 Abstract Syntax Trees

For the abstract syntax trees, we can find a similar instructions here at Week 4 Report for Abstract Syntax Trees. The rules again are listed as following:

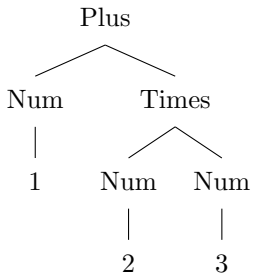
```
Plus.  Exp ::= Exp "+" Exp1 ;
Times. Exp1 ::= Exp1 "*" Exp2 ;
Num.   Exp2 ::= Integer ;

coercions Exp 2 ;
```

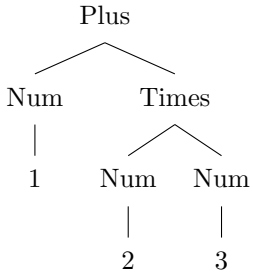
2+1:



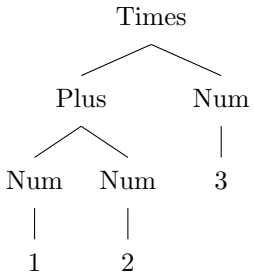
1+2*3:



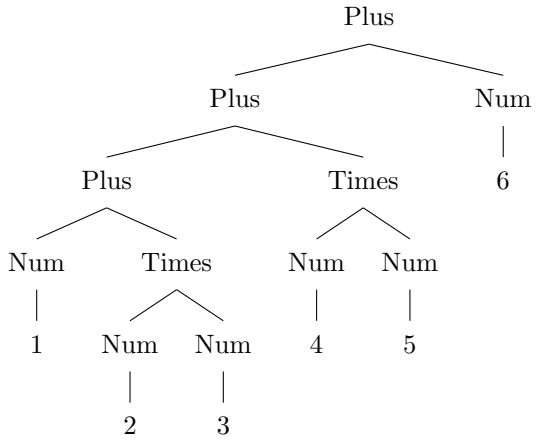
1+(2*3):



(1+2)*3



1+2*3+4*5+6:



In regards to the abstract syntax tree of 1+2+3, this would be identical to (1+2)+3 because we read operations from left to right which matches this order of parentheses.

2.5 Homework 5

Here are the evaluations found in Homework 5.

```
"x":
Prog (EVar (Id "x"))

"x x":
Prog (EApp (EVar (Id "x")) (EVar (Id "x")))

"x y":
Prog (EApp (EVar (Id "x")) (EVar (Id "y")))

"x y z":
Prog (EApp (EApp (EVar (Id "x")) (EVar (Id "y"))) (EVar (Id "z")))

"\ x.x":
Prog (EAbs (Id "x") (EVar (Id "x")))

"\ x.x x":
Prog (EAbs (Id "x") (EApp (EVar (Id "x")) (EVar (Id "x"))))

"(\ x . (\ y . x y)) (\ x.x) z":
Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EVar (Id "x")) (EVar (Id "y"))))) (EAbs (Id
    "x") (EVar (Id "x")))) (EVar (Id "z")))

"(\ x . \ y . x y z) a b c":
Prog (EApp (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EApp (EVar (Id "x")) (EVar (Id "y")))
    (EVar (Id "z"))))) (EVar (Id "a"))) (EVar (Id "b"))) (EVar (Id "c")))
```

The 2-D abstract syntax trees are found at the Github pdf here.
The abstract syntax trees for Homework 5 part 2 can be found at the pdf of the evaluations in the Github file.

3 Project

Introductory remarks ...

The following structure should be suitable for most practical projects.

3.1 Specification

3.2 Prototype

3.3 Documentation

3.4 Critical Appraisal

...

4 Conclusions

(approx 400 words)

In the conclusion, I want a critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of programming languages and software engineering?

References

[PL] Programming Languages 2022, Chapman University, 2022.