Language Proposal: “EqualsEquals”

Shortname: “eqeq”

Nam Hoang, nnh2110
Tianci Zhong, tz2278
Ruicong Xie, rx2119
Lanting He, lh2756
Jonathan Zacsh, jz2648

[Introduction](#)
[Motivation](#)
[Language Description](#)
- [Target Language: Python](#)
- [Syntax Overview](#)
  - [Data Types](#)
  - [Comments](#)
  - [Code block format](#)
  - [Declaration](#)
  - [Flow control](#)
- [Language Features](#)
  - [Mathematical Equations](#)
- [Sample program](#)

Introduction

EqualsEquals is a language designed for simple equation evaluation. EqualsEquals helps express mathematical equation in ASCII without straying too far from whiteboard-style notation. Users do not need to rearrange the equation by hand or reduce the formulas in order to get answers. It is similar to Wolfram Alpha that gives you information about an equation or expression. Our language can take multiple equations as input and it can evaluate the value of a certain variable of the equations when the values of other variables are given. It can also check if the set of equations have a valid solution. If not, it will throw exception and inform the user.

Motivation

Reducing mathematical formulas can be really painful and tedious. With our language you can type in the equations just as the way you write them down on paper, then the program will evaluate the equations and return the result for you. We want to simplify the process of evaluating equations. Our goal is that we can evaluate equations without manually defining

many programming-specific variables, so that the mathematical calculation procedure can be facilitated.

Language Description

Target Language: Python

Syntax Overview

Data Types

- `int`
- `double`
- **Vector**: `vector<number>` or `vector<vector>` numeric, random-access arrays of values
- **equations**: multiple variable declarations on a single line, showing set definition
    - (see Mathematical Equations below)
- **string** literals (*mainly for printing*)

Comments

Single and multi-line, C-style commenting is ignored by the compiler. Example:

```
// This is a comment

/* This
is
a
block
comment*/
```

Code block format

- Rely on curly braces to limit semantic scope (like Java/C)
- **Whitespace** is ignored
    - **Indentation** is insignificant whitespace

Declaration

Types are inferred where possible. For example:

```
g := 9.8 //infer that g is double
```

Flow control

- If statement; boolean evaluation of equations can be done directly, eg:
  - `if (a = b) {...}`
- equations are listed at the beginning, strictly followed by evaluation
- `evaluate` keyword to loop with limited scope
- Equal operators:
  - `:=` is the assignment operator. For example, `b := b - 1` decrements b by 1
  - `=` is the equal operator (in math). For example, `b = b - 1` does not make sense
- Arithmetic operators: `+ - * / ^`
  - note: multiplication is implied by spaces (see "Mathematical Equations")
- Built-in function
  - Math evaluation functions, eg: `sqrt()`, `log()`, `integrate()`, `differentiate()`, etc.

Language Features

Mathematical Equations

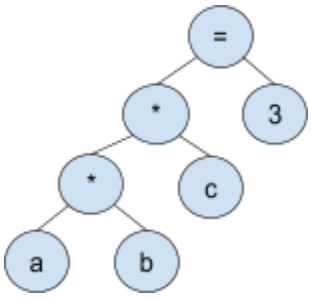
Equations are written as they are on whiteboard. No explicit simplification required. Cross-evaluation among equations are supported, eg:

"`a b c = 3`" implicitly declares three new variables, along with a equation:

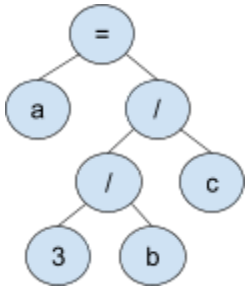
- `a = 3/(bc)`
- `b = 3/(ac)`
- `c = 3/(ab)`

Equations are stored internally as a binary expression tree to facilitate quick and easy mathematical manipulation. , eg:

"`a b c = 3`" is stored as a tree:



The expression is equivalent "`a = 3/b/c`" and the tree below:



Sample program

```
/** START: Example of equations' `evaluate` use-cases */

sum := 0  // init a number called sum
vecs<number>  // init a vector<number> called vecs
pendulum {
    /**
     * Give our compiler some equations
     */

    m * g * h = m * v^2 / 2  // inferred equation, because unknown: {m, v}
    h = l - l * cos(theta)  // inferred: equation, because `l` unknown
    // note: relying on existing libraries for cos

    /**
     * some set-theory here-- pass our compiler some subset of the unknowns
     */
    m = 10

    // .. potentially a lot of givens
}

// evaluate v in pendulum's equations given that g = 9.8 and l in range(20)
pendulum: evaluate l := range(0, 20) {
    g = 9.8

    // Our compiler now has solutions to: m, g, l (and indirectly h), so v can be solved:
    print("velocity: %d", v)  // v is automatically evaluated when it's referred to
}

// evaluate v in pendulum's equations given that g in range(4, 15) and l = 10
// take the average of values of v
pendulum: evaluate g := range(4, 15) {
    l = 10
    sum += v
    vecs.append(v)  // add v to vecs
}
average := sum / (15 - 4)
```

```
// Example: tries l = 10, v = 20 in context of pendulum, to see its equations are still true.
// If equations are inconsistent, the program will throw an exception.
pendulum: evaluate {
    l = 10
    v = 20

    catch Exception {
        print("bad")
    }
    print("good")
}

// Example of "global" evaluation (no access to the equations of `pendulum`)
evaluate {
    a = 4
    1 + 2 = a  // This throws an exception
}
/** END: Example of equations' `evaluate` use-cases */

/* START: Example of a function (multi-line equations) to find gcd of a and b */
myGCD {
    gcd = {
        if (0 = b) {
            return a
        } elif (a = 0) {
            return b
        }

        if (a > b) {
            a, b := b, a % b  // note: multiple assignments on single line
        } else {
            a, b := b % a, a
        }
        return gcd
    }
}

// evaluate gcd(10, 20)
myGCD: evaluate {
    a = 10
    b = 20

    print("gcd of %d and %d is %d", a, b, gcd)
}
/* END: Example of a function (multi-line equations) to find gcd of a and b */
```