EqualsEquals Language Reference Manual

name	email	role	UNI
Nam Nhat Hoang	nam.hoang@columbia.edu	System Architect	nnh2110
Tianci Zhong	tz2278@columbia.edu	Manager	tz2278
Ruicong Xie	rx2119@columbia.edu	Manager	rx2119
Lanting He	lh2756@barnard.edu	Tester	lh2756
Jonathan Zacsh	jz2648@columbia.edu	Language Guru	jz2648

Introduction

EqualsEquals - "eqeq" for short - is a language designed for simple equation evaluation. EqualsEquals helps express mathematical equation in ASCII (though UTF-8 string literals are allowed) without straying too far from whiteboard-style notation. Users do not need to be overly careful to perfectly reduce formulas behind. Leaving unknowns for later is possible, without encapsulating equations' unknowns as function parameters. EqualsEquals takes multiple equations as input and can evaluate the value of a certain variables of the equations when the values of other variables are given.

Motivation

Reducing mathematical formulas can be really painful and tedious. We want to simplify the process of evaluating equations. With our language we take a step to help users leave their formula in a similar format to what they'd normally have on paper.

Overview

Valid source programs will compile down to C.

Definition of a Program

The simplest - though contrived - valid program is:

```
find { printf("Hello, all %.0f readers!\n", 21 * 2); }
```

Which prints the following to standard out: `Hello, all 42 readers!`

Formally, a valid program is a series of:

- one or more `find` blocks.
- zero or more "context" blocks *(aside from the automatic, global context)*

"Context"s & `find` Blocks

While both types of blocks of code are simply curly brace enclosed listings of sequential statements, contexts and `find` blocks differ in their use:

- **context**s are expected to layout and define equations for use later. Thus they're allowed semantic gaps in their equations; eg: missing solutions.
- `find` blocks on the other hand are expected to be the resolution to "find" missing said pieces, or simply apply completed solutions to new inputs.

It follows then that `find` expressions *apply* to contexts. Where a context might be shared for re-use, `find` expressions are designed to make local use of equations in a given context.

Though the above "Hello World" example executes a `find` on the global context, users will generally define contexts manually. For example a "Euclid" context, where `gcd` might be defined:

```
Euclid = { gcd = /*... defined here ...*/; }
Euclid:find gcd {
  a = 20; b = 10; print("%.0f\n", gcd);
}
```

Design Implementation

Within contexts and `find` blocks, valid statements look like many C-style languages, where expressions are semi-colon (`;`) separated, may be have sub-expressions using parenthesis (`(`, `)`) and the lexemes of an expression may be made up of:

1. **variables** to which floating-point numbers are assigned
    - **vectors**, like variables, but have square brackets (`[]`) after their identifier is indeed a *vector* of numbers, eg: `myVector[]`
2. **arithmetic** expressions: addition, subtraction, multiplication, division, exponents
3. **comments** characters ignored by the compiler
4. **white-space** to arbitrary length (eg: `a = 3` is the same as `a  = 3`)
5. **string** literals used for printing
6. **equality** operations in `if`/`else` expressions *(which evaluate to `1` or `0` if both operators are equal)*

Tokens: Expressions' Lexemes

Below is the syntax of each type of expression. For the semantic description of each, refer to the "Declarations" section, below.

1. Floating point numbers, including integers:

   eg: `123`, `1.34e-4`, `0.23`, `.13`, `0e1`.

   Described by the regular expression `flt` here:

   ```
   let pos = ['1' - '9']                    in
   let dig = '0' | pos                      in
   let exp = ('e' | 'E') ('-' | '+')? pos+  in
   let fra = '.' dig+ exp?                  in
   let num = pos dig*                       in
   ```

```
let flt = num | ((num | 0)? fra) | (num exp)
```

2. Variable Assignment: numbers stored with user-defined names:

   eg: `weight = 100 /*grams*/`

   Described by the regular expression `var` here:

```
let aph = ['a'-'z'] | ['A'-'Z']     in

let var = aph+ ('_' | ['0'-'9'])*
```

3. Contexts: blocks of symbols:

   eg: `Euclid: {/* any number of lines of EqualsEquals here */}`

   Building on variables' definition, the regular expression can be described by `ctx` here:

```
let ctx = ['A'-'Z'] var*
```

4. Strings: mostly used for printing results:

   eg: `printf("result of my maths: %.0f\n", gcd)`

   String literals can be described by the regular expression `str` here:

```
let chr = \x(0...9|A...F|a...f)(0...9|A...F|a...f) in
let spc = \(\n| \t| \b| \r| ' ')
let num = ['0' - '9']                   in
let aph = ['a' - 'z'] | ['A' - 'Z']     in

let str = (aph | num | chr | spc)*
```

Reserved Keywords

Following are reserved keywords, and have special meaning in the language. See "Statements" and "Declarations" sections elsewhere for each of their meanings.

- `if`
- `elif`
- `else`
- `find`
- `print`

Declarations

1. A list of declarator are separated by comma. Formatted as below:

```
Declarator-list:
declarator, declarator, ...
```

   For example:

```
a = 2, b = 3;
a = b, b = a % b;
```

2. Variable:

   To declare a variable, only name of the variable is needed. The data types of the variables are inheritable.

   Possible inherited data types:

   - Double
   - String

3. Vector:

```
V[ ]
V[constant-expression]
V = {a, b, c, ...}
```

   In the first case, the expression will declare an array with length 1 and initialized with zero, as `[ 0 ]`. In the second case, the expression will declare an array with length that evaluated result of the constant expression and initialized with zeros, as `[ 0, 0, ... , 0]`. The constant expression need to be evaluated to an integer. Such a declarator makes the contained identifier have type `vector`. The declarator `V[ i ]` yields a 1-dimensional array with rank i of objects of type `double`. To declare a vector of vectors, the notation would be like `V[i][j]`. In the third case, the expression will declare an array with length, the number of elements inside the "{}". It will initialize the array with the elements in the "{}". The elements have to be either Double or String and could not be fixed of both.

4. Multi-line equation: declaration of multi-line equation has the format:

```
equation_name = {
 // some operations
 var; // a variable, indicating equation_name's value
}
```

The `equation_name` has the type Double, where `var` indicates the name of variable expression holding the desired value. The equation will be passed by value. The multi-line equations, like regular equations, can only express one value *(or a vector of values)*.

For example:

```
gcd = {
  if (0 == b) {
    a;  // solution is a
  } elif (a == 0) {
    b;  // solution is b
  }

  if (a > b) {
    a = b, b = a % b;
    // note: multiple assignments on single line
  } else {
    a = b % a, b = a;
  }
  gcd; // solution is gcd w/the current a and b
}
```

This example results in an expression `gcd` - similar to a C-style function - that can be referred to later, given the necessary inputs `a` and `b` (in eqeq's case, the right "context").

5. Equations:

```
variable = variable (value assigned?)
variable = some arithmetic expression
variable = { /*some multi-line equation that evaluates to a number*/ }
```

Only variable will be allowed on the left side of the equal sign. The expression on the right side can be a declared variable, an arithmetic expression that evaluates to a number, or a multi-line equation enclosed in curly-braces (see "Multi-line equation" above).

For example:

```
a = 3; b = a;           // b == 3
a = 3; b = a * 2 + 1;   // b == 7
a = 3; b = 6; c = gcd; // c == 3
```

For analysis of equation arithmetic, see "Expression Precedence & Meaning", below.

6. Scopes (access to variables):

```
VAR = EXPR

Scope_name {
 list of equations

 // VAR = EXPR // overwrites global `VAR`
}

Scope_name: find VAR [with VAR_B in range()]* ] {
 /** code here has access to `Scope_name`'s equations */
}
```

Here, `Scope_name` is like an object of equations. Equations are put inside the bracket follow `Scope_name`.

Any variable declared outside of a `Scope_name` is a global variable that can be accessed from anywhere within the program. If a variable declared in some `Scope_name` has the same name as some global variable, it will overwrite the value within the `Scope_name`. After getting out of the `Scope_name`, the variable will restore its value.

`Scope_name: find VAR [...]` is the evaluation part. A `with` clause is optional. See "With Statement" section below. `find` will evaluate the variable following it using the equations inside the `Scope_name` block. Once a `Scope_name` is defined, multiple `find` are allowed to use the equations inside it.

Statements

Expression Statement

Expression statements are statement that includes an expression and a semicolon at the end:

```
expression ;
```

Combining Statements

A statement can be the multiple of other statements. `{` and `}` are used to group multiple statements as one statement. So the form of compound statements is:

```
{ statement+ }
```

, which means that a compound statement has an opening curly bracket, one or more statements, and a closing curly bracket.

Conditional Statement

Statements that are used in conditional statements:

```
// if_statement
if ( expression ) statement

// elif_statement
elif ( expression ) statement

// else_statement
else statement
```

Conditional statements have the following form:

```
if_statement elif_statement* else_statement?
```

, which means that it contains a required if_statement, any number of elif_statement, and an optional else_statement.

While Statement

While statements have the form:

```
while ( expression ) statement
```

The sub-statement is executed repeatedly so long as the value of the expression remains non-zero.

Break Statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, or `with` statement.

Continue Statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while` or `with` statement; that is to the end of the loop. More precisely, in each of the statements.

Context statement

A context statement include a context name and a compound statement:

```
context_name compound_statement
```

To access a context, we use a statement with the following form:

```
context_name: statement
```

The sub-statement will be evaluated in the context given by `context_name`.

Examples:

```
mycontext {
  x = 5;
}

print(x); // throw an exception because x in not defined
mycontext: find x {
  print(x); // prints 5
}
```

With Statement

With statements have the form

```
with [variable in expression, ]+ compound_statement
```

, which means that with takes one or more expressions, and a compound sub-statement.

If the expressions have type double, then with will evaluate the expression and execute the compound sub-statement:

```
with x in 5 {
  print(x);
}  // 5

with x in 5, y in 6 {
  print(x + y);
}  // 11
```

If the expressions have type vectors, we will execute the compound sub-statement with all the combinations of values available. Basically, it mirrors multiple `for` loop in Python:

```
// with vector assignment (causing equivalence of `for` loop in other languages)
with x in {1, 2, 3} {
  print(x);
}  // print 1, 2, 3 on 3 separate lines

with x in {1, 2}, y in {4, 6} {
  print(x, y);
}  // print 5, 7, 6, 8 on 4 separate lines
```

Find Statement

Find statements start with keyword `find` and an expression, followed by a sub-statement:

```
find expression statement
```

In a find statement, the last statement should be evaluated with access to previously declared expressions.

Examples of find statements:

```
// a simple example
velocity = length + 1;

find velocity {
  length = 5;
  print(velocity);
} // print 6

// this block is the same as the one above
find velocity with length in 5 {
  print(velocity);
} // print 6

pendulum:find vector with length in range(0, 5) {
  print(velocity);
} // print 1 to 6
```

Built-ins

`print()`

`print()` is built-in function that mirrors the C `printf()` API. `print()`'s arguments include a string, and optional expressions:

```
print( a_string_with_formatters [, expressions]* )
```

`print()` prints the formatted string to the screen.

Users can format strings in `print()` with `%f` and `%s` formatter (and but not `%d`, since `eqeq` only uses float). For example,

```
print("words here %f.0 and %f here\n", 4, myvar);
// words here 4 and 3.14159 here
```

`range()`

`range()` mimics Python's `range()` function. It takes an optional expression `start`, an expression `stop`, and an optional expression `step`. It returns a vector from `stat` to `stop - 1`, with distance `step` between each member of the vector:

```
range([start,] stop [,step]);
```

For examples,

```
range(3);        // same as writing: {0, 1, 2}
range(2, 5);     // same as writing: {2, 3, 4}
range(2, 8, 3);  // same as writing: {2, 5, 8}
```

Expression Precedence & Meaning

Here various expressions' meanings are described, generally shown as `expr`, in the order of their precedence.

- `'(' expr ')'`: for sub-expressions. For example, `expr` of `4 + 5` here:

  ```
  b * (4 + 5); // `expr` should be considered first
  b *  9;      // same as above; note absence of parenthesis
  ```

- `id '[' expr? ']'`: for vector access.
- `-expr`: negative. The result is the negative of the expression. Note, the type of the expression must be double.
- `!expr`: logical negation.

  The result of the logical negation operator `!` is `1` if the value of `expr` is `0`. If the value of `expr` is anything other than `0`, then `!expr` results in `0`.
- `left_expr ^ right_expr`: exponentiation. Mathematically raises `left_expr` to the power, `right_expr`. Note: uses underlying C standard library's corresponding power API, eg: `double pow (double base, double power).`
- `expr * expr`, `expr / expr` The binary operator * / indicates multiplication and division operation. If both operands are double, the result is double.
- `expr % expr` The binary `%` operator yields the remainder from the division of the first expression by the second. Both operands are double, and only integer portion of the double will be used for modular operation, and the result is a double with fraction equals to zero. eg:

  ```
  12.0 % 7.0 = 5.0;
  12.3 % 7.5 = 5.0;
  ```
- `expr + expr`, `expr - expr` The result is the sum or different of the expressions. Both are double, the result is double.
- equality/inequality:
  - `expr > expr`, `expr >= expr`, `expr < expr`, `expr <= expr` The operators `<` (less than), `>` (greater than), `<=` (less than or equal to) and `>=` (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the `+` operator.
  - `expr != expr`, `expr == expr`: The `!=` (not equal to) and the `==` (equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a < b == c < d` is `1` whenever `a < b` and `c < d` have the same truth-value).
- `expr || expr` The `||` operator returns 1 if either of its operands is non-zero, and 0 otherwise. It guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.
- `expr && expr` The `&&` operator returns 1 if both of its operands is non-zero, and 0 if either is 0. It guarantees left-to-right evaluation; moreover, the second

operand is not evaluated if the value of the first operand is 0.

- `left_expr = right_expr`: assignment. the `left_expr` must be a single variable expression. The result of this operation is that `left_expr` holds the value of `right_expr` going forward. If `right_expr` contains unknown variables, the `left_expr` will not be solvable until a `find` block expresses a solution in terms of `left_expr` and provides any missing variables from the `right_expr`.

- `expression , expression` A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand.

Sample program

Below are example programs in EqualsEquals.

Example of equations' `find` Use-cases

```
sum = 0  // initialize a number called sum
pendulum {
  /**
   * Spell out equation for our compiler:
   *   m * g * h = m * v^2 / 2
   */
  m = 10;
  theta = pi / 2;
  g = 9.8;
  h = l - l * cos(theta); // cosine, being a built-in
  v = (2 * g * h) ^ (1 / 2); // square root
  // note: relying on existing libraries for cos
}

// evaluate v in pendulum's equations given that g = 9.8 and l in range(20)
pendulum: find v with l in range(0, 20) {
  // 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: find v with g in range(4, 15), m = 100 {
  l = 10;

  sum += v;
  // scope of sum: global (b/c it's not in the scope of pendulum but would be
  // overwritten by pendulum)
}

average = sum / (15 - 4);

pendulum: find v with v in range(20) {
  // throw a compiler error because can't find v with v's value
}

// 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: find v {
  l = 10; // by now, v will be calculated
  print(v == 20);  // print 1
  v = 20; // throws an error
}
```

Example of a multi-line equation to find `gcd` of `a` and `b`

```
myGCD {
  gcd = {
    if (0 == b) {
      a;  // solution is a
    } elif (a == 0) {
      b;  // solution is b
    }

    if (a > b) {
      a = b, b = a % b;
      // note: multiple assignments on single line
    } else {
      a = b % a, b = a;
    }
    gcd; // solution is gcd w/the current a and b
  }
}

// evaluate gcd of 10 and 20
myGCD: find gcd {
  a = 10;
  b = 20;

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


/* This works too. In this case, gcd is not in any special scope */
gcd = {
  ...  // same as the above example
}

// evaluate gcd of 10 and 20
find gcd {
  a = 10;
  b = 20;
  print("gcd of %d and %d is %d", a, b, gcd);
}
/* END: Example of a multi-line equations to find gcd of a and b */
```