CAT Language Specification

Michael Faulkner

May 20, 2024

Introduction

CAT (Calculatron and Algebra Tutor) is a domain-specific language designed to assist individuals with mathematical calculations. More specifically, the language allows users to input mathematical expressions and operations to perform on the expressions. Running a program in this language will then produce a textual represenation of the ensuing calculation as well as a LaTeX file that can be compiled into a PDF that displays the calculation in a pretty manner.

While there exist computational tools for generalized mathematical expressions (for example Mathematica), such tools tend to focus only on getting to an answer. The purpose of CAT is to assist its users in understanding *how* to get the answer by outputting the intermediate steps of the calculation. This addresses another point of annoyance with tools like Mathematica that will sometimes output, seemingly arbitrary, restrictions on the parameters alongside its final expression. By working through the intermediate steps of the calculation, CAT should allow its users to understand why such restrictions were imposed and what the limits of the calculation would be if such assumptions were not made.

Video Presentation

View the video here for a video explanation of CAT! If the above link doesn't work you can type the following into your browser: https://youtu.be/nhKoWXYoXG4

Design Principles

The primary goal of CAT is to produce easily understandable calculations. This goal corresponds with a prioritization of simpler techniques that are easier to follow over more powerful techniques that might be able to find more general answers but are less decipherable. This likely also means a tradeoff in terms of speed as it is necessary to work through the intermediate details of a calculation sufficiently to make them human readable.

Examples

The following examples are included in a folder called `examples`. You can run these examples using the interpreter. For example, to run example 1, you can use the command `dotnet run examples/example1.cat` at the project level. Two things will happen when you run a program: the simplification process will be printed out as textual output, and a LaTeX file will be created in the same directory with the corresponding name (i.e. running the above command will produce the file `example1.tex` in the examples directory). You can compile the LaTeX file into a pdf using whatever your standard LaTeX compiler is (for example, pdflatex).

1. The following example illustrates how each line of a CAT program is a separate expression to be simplified. The first line here simplifies to simplify to 2, and the second line simplifies to -1 .

   Program:

   ```
   ((1 + 3^2) / 2 - 1)^(1/2)
   (x+1)(x-1) - x^2
   ```

Textual Output:

```
Expanding:  ((1 + 3^2)2^(-1) + (-1)1)^((1)2^(-1))
==> ((-1)1 + (1 + 3^2)2^(-1))^((1)2^(-1))
==> ((-1)1 + (1 + (3)3)2^(-1))^((1)2^(-1))
==> ((-1)1 + (1)2^(-1) + 3(3)2^(-1))^((1)2^(-1))
Simplifying:  ((-1)1 + (1)2^(-1) + 3(3)2^(-1))^((1)2^(-1))
==> ((-1)1 + (1)2^(-1) + 3(3)2^(-1))^((1)2^(-1))
==> (-1 + (1)2^(-1) + 3(3)2^(-1))^((1)2^(-1))
==> (-1 + 3(3)2^(-1) + 2^(-1))^((1)2^(-1))
==> (-1 + 0.5 + 3(3)2^(-1))^((1)2^(-1))
==> (-1 + 0.5 + 0.5(3)3)^((1)2^(-1))
==> (-1 + 0.5 + 4.5)^((1)2^(-1))
==> 4^((1)2^(-1))
==> 4^2^(-1)
==> 4^0.5
==> 2

Expanding:  (x + 1)(x + (-1)1) + (-1)x^2
==> (1 + x)(x + (-1)1) + (-1)x^2
==> (-1)(1)1 + (-1)1x + 1x + xx + (-1)x^2
==> (-1)(1)1 + (-1)1x + (-1)xx + 1x + xx
Simplifying:  (-1)(1)1 + (-1)1x + (-1)xx + 1x + xx
==> (-1)(1)1 + (-1)1x + (-1)xx + 1x + xx
==> -1 + (-1)1x + (-1)xx + 1x + xx
==> -1 + (-1)x + (-1)xx + 1x + xx
==> -1 + (-1)x + 1x + xx + (-1)x^2
==> -1 + x + (-1)x + xx + (-1)x^2
==> -1 + x + (-1)x + (-1)x^2 + x^2
==> -1 + 0 + 0
==> -1 + 0
==> -1
```

LaTeX Output:

Expression: $((1 + 3^2)2^{-1} + (-1)1)^{(1)2^{-1}}$

Expanding: $((1 + 3^2)2^{-1} + (-1)1)^{(1)2^{-1}}$

$$((-1)1 + (1 + 3^2)2^{-1})^{(1)2^{-1}}$$

$$((-1)1 + (1 + (3)3)2^{-1})^{(1)2^{-1}}$$

$$((-1)1 + (1)2^{-1} + 3(3)2^{-1})^{(1)2^{-1}}$$

Simplifying: $((-1)1 + (1)2^{-1} + 3(3)2^{-1})^{(1)2^{-1}}$

$$((-1)1 + (1)2^{-1} + 3(3)2^{-1})^{(1)2^{-1}}$$

$$(-1 + (1)2^{-1} + 3(3)2^{-1})^{(1)2^{-1}}$$

$$(-1 + 3(3)2^{-1} + 2^{-1})^{(1)2^{-1}}$$

$$(-1 + 0.5 + 3(3)2^{-1})^{(1)2^{-1}}$$

$$(-1 + 0.5 + 0.5(3)3)^{(1)2^{-1}}$$

$$(-1 + 0.5 + 4.5)^{(1)2^{-1}}$$
$$4^{(1)2^{-1}}$$
$$4^{2^{-1}}$$
$$4^{0.5}$$
$$2$$

Expression: $(x + 1)(x + (-1)1) + (-1)x^2$

Expanding: $(x + 1)(x + (-1)1) + (-1)x^2$

$$(1 + x)(x + (-1)1) + (-1)x^2$$
$$(-1)(1)1 + (-1)1x + 1x + xx + (-1)x^2$$
$$(-1)(1)1 + (-1)1x + (-1)xx + 1x + xx$$

Simplifying: $(-1)(1)1 + (-1)1x + (-1)xx + 1x + xx$

$$(-1)(1)1 + (-1)1x + (-1)xx + 1x + xx$$
$$-1 + (-1)1x + (-1)xx + 1x + xx$$
$$-1 + (-1)x + (-1)xx + 1x + xx$$
$$-1 + (-1)x + 1x + xx + (-1)x^2$$
$$-1 + x + (-1)x + xx + (-1)x^2$$
$$-1 + x + (-1)x + (-1)x^2 + x^2$$
$$-1 + 0 + 0$$
$$-1 + 0$$
$$-1$$

2. This second example highlights CAT's ability to deal with simplifying deeply nested expressions and ultimately undwinding all of the complexity to get a simple answer. Here, we have an expression of the form $1 - x^z$ where z is a complicated expression. But z ultimately evaluates to 0, so the overall expression will evaluate to 0.

   Program:
   ```
   1 - x^(1-x^2/((x + 1)(x - 1) + 1))
   ```

   Textual Output:
   ```
   Expanding:  1 + (-1)x^(1 + (-1)((x^2)((x + 1)(x + (-1)1) + 1)^(-1)))
   ==> 1 + (-1)x^(1 + (-1)(x^2)(1 + (1 + x)(x + (-1)1))^(-1))
   ==> 1 + (-1)x^(1 + (-1)xx(1 + (1 + x)(x + (-1)1))^(-1))
   ==> 1 + (-1)x^(1 + (-1)xx(1 + (-1)(1)1 + (-1)1x + 1x + xx)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xx(1 + (-1)(1)1 + (-1)1x + 1x + xx)^(-1))
   Simplifying:  1 + (-1)(x^1)x^((-1)xx(1 + (-1)(1)1 + (-1)1x + 1x + xx)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xx(1 + (-1)(1)1 + (-1)1x + 1x + xx)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xx(-1 + 1 + (-1)1x + 1x + xx)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xx(-1 + 1 + (-1)x + 1x + xx)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xx(-1 + 1 + x + (-1)x + xx)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xx(-1 + 1 + x + (-1)x + x^2)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xx(-1 + 0 + 1 + x^2)^(-1))
   ==> 1 + (-1)(x^1)x^((-1)xxx^2^(-1))
   ```

```
==> 1 + (-1)(x^1)x^((-1)x^2^(-1 + 1))
==> 1 + (-1)(x^1)x^((-1)x^2^0)
==> 1 + (-1)(x^1)x^((-1)1)
==> 1 + (-1)(x^(-1))x^1
==> 1 + (-1)x^(-1 + 1)
==> 1 + (-1)x^0
==> 1 + (-1)1
==> -1 + 1
==> 0
```

LaTeX Output:

Expression: $1 + (-1)x^{1+(-1)(x^2((x+1)(x+(-1)1)+1)^{-1})}$

Expanding: $1 + (-1)x^{1+(-1)(x^2((x+1)(x+(-1)1)+1)^{-1})}$

$$1 + (-1)x^{1+(-1)x^2(1+(1+x)(x+(-1)1))^{-1}}$$

$$1 + (-1)x^{1+(-1)xx(1+(1+x)(x+(-1)1))^{-1}}$$

$$1 + (-1)x^{1+(-1)xx(1+(-1)(1)1+(-1)1x+1x+xx)^{-1}}$$

$$1 + (-1)x^1x^{(-1)xx(1+(-1)(1)1+(-1)1x+1x+xx)^{-1}}$$

Simplifying: $1 + (-1)x^1x^{(-1)xx(1+(-1)(1)1+(-1)1x+1x+xx)^{-1}}$

$$1 + (-1)x^1x^{(-1)xx(1+(-1)(1)1+(-1)1x+1x+xx)^{-1}}$$

$$1 + (-1)x^1x^{(-1)xx(-1+1+(-1)1x+1x+xx)^{-1}}$$

$$1 + (-1)x^1x^{(-1)xx(-1+1+(-1)x+1x+xx)^{-1}}$$

$$1 + (-1)x^1x^{(-1)xx(-1+1+x+(-1)x+xx)^{-1}}$$

$$1 + (-1)x^1x^{(-1)xx(-1+1+x+(-1)x+x^2)^{-1}}$$

$$1 + (-1)x^1x^{(-1)xx(-1+0+1+x^2)^{-1}}$$

$$1 + (-1)x^1x^{(-1)xx(x^2)^{-1}}$$

$$1 + (-1)x^1x^{(-1)(x^2)^{-1+1}}$$

$$1 + (-1)x^1x^{(-1)(x^2)^0}$$

$$1 + (-1)x^1x^{(-1)1}$$

$$1 + (-1)x^{-1}x^1$$

$$1 + (-1)x^{-1+1}$$

$$1 + (-1)x^0$$

$$1 + (-1)1$$

$$-1 + 1$$

$$0$$

3. This last example highlights CAT's current ability to factor expressions. CAT automatically expands expressions before simplifying them, so this example demonstrates how sometimes CAT will not recover the original expression. In a perfect world, this expression would simplify to `(x + y + z)^2`. But after first choosing to factor out the two, CAT gets stuck trying to simplify this expression any further.

Program:
```
abcd + bdef
(x+y+z)^2
```

Textual Output:
```
Expanding:  abcd + bdef
==> abcd + bdef
Simplifying:  abcd + bdef
==> abcd + bdef
==> bd(ac + ef)

 Expanding:  (x + y + z)^2
==> (x + y + z)^2
==> (z + y + x)(z + y + x)
==> xx + xy + xy + yy + xz + xz + yz + yz + zz
Simplifying:  xx + xy + xy + yy + xz + xz + yz + yz + zz
==> xx + xy + xy + yy + xz + xz + yz + yz + zz
==> xy + xy + yy + xz + xz + yz + yz + zz + x^2
==> xy + xy + xz + xz + yz + yz + zz + x^2 + y^2
==> xy + xy + xz + xz + yz + yz + x^2 + y^2 + z^2
==> 2xy + 2xz + 2yz + x^2 + y^2 + z^2
==> 2(xy + xz + yz) + x^2 + y^2 + z^2
==> 2(yz + x(y + z)) + x^2 + y^2 + z^2
```

LaTeX Output:

Expression: $abcd + bdef$

Expanding: $abcd + bdef$

$$abcd + bdef$$

Simplifying: $abcd + bdef$

$$abcd + bdef$$

$$bd(ac + ef)$$

Expression: $(x + y + z)^2$

Expanding: $(x + y + z)^2$

$$(x + y + z)^2$$

$$(z + y + x)(z + y + x)$$

$$xx + xy + xy + yy + xz + xz + yz + yz + zz$$

Simplifying: $xx + xy + xy + yy + xz + xz + yz + yz + zz$

$$xx + xy + xy + yy + xz + xz + yz + yz + zz$$

$$xy + xy + yy + xz + xz + yz + yz + zz + x^2$$

$$xy + xy + xz + xz + yz + yz + zz + x^2 + y^2$$
$$xy + xy + xz + xz + yz + yz + x^2 + y^2 + z^2$$
$$2xy + 2xz + 2yz + x^2 + y^2 + z^2$$
$$2(xy + xz + yz) + x^2 + y^2 + z^2$$
$$2(yz + x(y + z)) + x^2 + y^2 + z^2$$

Language Concepts

At it's core, CAT is a way of expressing and manipulating mathematical expressions. Each line of a CAT program constitutes an expression that will be evaluated. Evaluating an expression involves reducing it to the simplist form possible by applying operators and performing algebraic manipulations. The primitives of CAT are variables (i.e. x, y, z) and numbers (i.e. $1, -10, 3.141592$). These primitives can be combined with mathematical operations such as +, -, *, /, and ^ in order to form expressions. Each CAT program must end with a `.cat` extension, and there must not be any trailing newlines or the parser will be upset that you are asking it to simplify an empty expression.

When you run a program in the interpreter, the program will print out the series of simplifications that it performs on each expression you gave it. CAT begins simplifying an expression by first expanding it, so that terms are fully distributed. Then, CAT begins combining terms and will attempt to refactor the remaining expression into the simplist form it can. In addition to printing these steps out to `stdout`, the interpreter will also create a LaTeX file with the same name as the input program containing another represenation of the calculation that is slightly easier to read. The LaTeX file should be easily compiliable using your prefered LaTeX compiler.

Formal Syntax

```
<instruction>  ::=  <expr> ('\n' <expr>)*
<expr>         ::=  <ws>*<operation><ws>*
                  |  <ws>*<parens><ws>*
                  |  <ws>*<literal><ws>*
<operation>    ::=  <expr><op><expr>
                  |  <expr><ws>+<expr>
                  |  <expr><parens>
                  |  <parens><expr>
                  |  -<expr>
<op>           ::=  + | - | * | / | ^
<parens>       ::=  (<expr>)
<literal>      ::=  <number>
                  |  <variable>
<number>       ::=  <digits>
                  |  -<digits>
                  |  <digits>.<digits>
                  |  -<digits>.<digits>
                  |  .<digits>
                  |  -.<digits>
                  |  <digits>.
                  |  -<digits>.
<digits>       ::=  <d><digits>
                  |  <d>
<d>            ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<variable>     ::=  a | b | c |...| x | y | z
<ws>           ::=  Any non-newline whitespace character.
```

Semantics

Syntax	Abstract Syntax	Prec/Assoc	Meaning
`<number>`	Number of double	N/A	`number` is a primitive. I represent numbers using the F# double datatype.
`<variable>`	Variable of char	N/A	`variable` is a primitive. I represent variables using the F# char datatype. These represent symbolic variables that are used in mathematical expressions.
`<expr> ('\n' <expr>)*`	Sequence of Expression list	0 / left	`Sequence` consists of a list of expressions to be simplified. Every CAT program has one `Sequence` which consists of each line of the program. After evaluating, each expression within a `Sequence` is converted into a list of expressions that represent progressive simplifications of the original expression.
`<expr> (+ <expr>)+`	Addition of Expression list	1 / left	`Addition` consists of a list of expressions to be added. Because addition is associative, the addends are stored in a list since their particular order doesn't matter.
`<expr> (- <expr>)+`	Addition of Expression list	1 / left	`Addition` also represents subtraction. Each term that is subtracted is represented with the addition of the term multiplied by negative one.
`<expr> (* <expr>)+`	Multiplication of Expression list	2 / left	`Multiplication` consists of a list of expressions to be added. Because multiplication is associative, the factors are stored in a list since their particular order doesn't matter.
`<expr>(<ws>+<expr>)*`	Multiplication of Expression list	2 / left	An alternative syntax for `Multiplication`. When two terms are next to each other they are implicitly multiplied.
`<expr><parens>`	Multiplication of Expression list	2 / left	An alternative syntax for `Multiplication`. When two terms are next to each other they are implicitly multiplied.
`<parens><expr>`	Multiplication of Expression list	2 / left	An alternative syntax for `Multiplication`. When two terms are next to each other they are implicitly multiplied.
`-<expr>`	Multiplication of Expression list	2 / N/A	An alternative syntax for `Multiplication`. In this case we have the implicit multiplication of the expression by negative one.
`<expr> (/ <expr>)+`	Multiplication of Expression list	2 / left	`Multiplication` also represents division. Each term that is divided is represented with the multiplication of the term by the multiplicative inverse of the other term (The multiplicative inverse is the number raised to the negative 1 power).

`<expr>^<expr>`	Exponentiation of Expression * Expression	3 / right	`Exponentiation` consists of two expressions. One represents the base, and the other represents the exponent. Note that Exponentiation is not stored as a list because it is not associative.

Remaining Work

Symbolic math systems can have nearly infinite scope. CAT has traded off a focus on computational power for its clarity in communicating its simplification steps. While there are essentially limitless additions that can be made to a computer math system, factoring expressions is clearly a current weakpoint of CAT. General purpose factoring of expressions is a very hard problem to solve given all the possible edge cases. Another goal that I had for CAT was to implement Calculus operations; however, one barrier I hit with trying to implement derivatives is that taking the derivative of arbitrary exponential functions requires being able to express logarithms, which brings us to the last notable area that CAT could be improved. CAT does not support special functions such as $\log$, $\sin$, $\sinh$, $\tan^{-1}$, $\exp$, etc. These functions are highly useful (especially in the context of calculus), so they logically be some of the next things that should be added to the language as well.