Lazy Math Instructor

Proseminar: Selected Fun Problems of the ACM Programming Contest

Ludwig Kolesch

University of Tübingen

July 14, 2023

Introduction to the problem

The lazy math instructor needs to check arbitrary terms for equivalence.
Example term:

```
(3*a) - c + (b*b)*8
```

Introduction to the problem

The lazy math instructor needs to check arbitrary terms for equivalence.
Example term:

Integers

- ▷ non-negative
- ▷ (single-digit)

`(3*a) - c + (b*b)*8`

Introduction to the problem

The lazy math instructor needs to check arbitrary terms for equivalence.
Example term:

Integers
- ▷ non-negative
- ▷ (single-digit)

Variables
- ▷ a to z

```
(3*a) - c + (b*b)*8
```

Introduction to the problem

The lazy math instructor needs to check arbitrary terms for equivalence.
Example term:

Integers
- non-negative
- (single-digit)

Variables
- a to z

```
(3*a) - c + (b*b)*8
```

Brackets
- matching
- can be nested

Introduction to the problem

The lazy math instructor needs to check arbitrary terms for equivalence.
Example term:

Integers
- non-negative
- (single-digit)

Variables
- a to z

```
(3*a) - c + (b*b)*8
```

Brackets
- matching
- can be nested

Operators
- +, − and *
- equal precedence

Introduction to the problem (2)

The lazy math instructor needs to check arbitrary terms for equivalence.

Integers
- ▷ non-negative
- ▷ (single-digit)

Variables
- ▷ a to z

`(3*a) - c + (b*b)*8`

Brackets
- ▷ matching
- ▷ can be nested

Operators
- ▷ +, - and *
- ▷ equal precedence

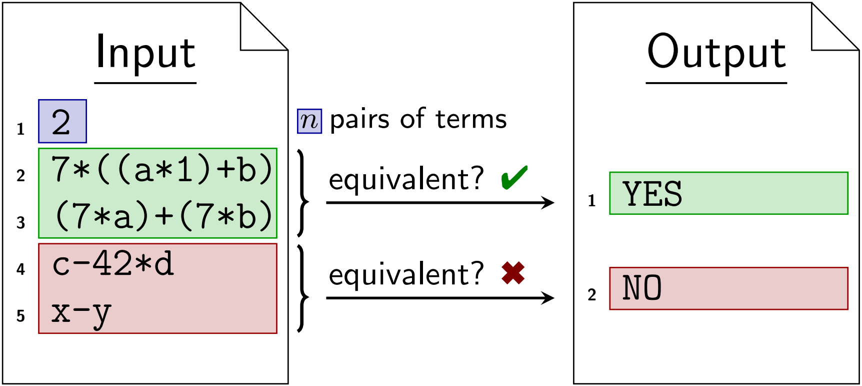
More example terms:

- `c - 42 * d` (left to right evaluation $\Rightarrow (c - 42) * d$)
- `7 * ((a*1)+b)`
- `(7*a) + (7*b)`

$\left.\begin{array}{l}\texttt{7 * ((a*1)+b)}\\ \texttt{(7*a) + (7*b)}\end{array}\right\}$ equivalent

Problem specification

Input: number n and $2n$ terms (n pairs)
Output: n lines: "YES" or "NO" to the equivalence of each pair



Plan of attack

How to check arbitrary terms for equivalence?

- Normalize the term into a polynomial (e.g. $3a^2bd - 6b^2c^5 + 42$)
- Check if the resulting polynomials are identical
  - Unique ordering of sub-terms inside a polynomial

Plan of attack

How to check arbitrary terms for equivalence?

- Normalize the term into a polynomial (e.g. $3a^2bd - 6b^2c^5 + 42$)
- Check if the resulting polynomials are identical
  - Unique ordering of sub-terms inside a polynomial

Example:

Term:	`(b * a * b) - (3 * 7) - (a * b * b) + (a * a * a)`
Parsed:	$a^3 + -21$
Term:	`(a * a * a) - 21`
Parsed:	$a^3 + -21$

Parsed polynomials are identical $\iff$ Terms are equivalent.

Polynomial representation

Example polynomial: $3a^2bd - 6b^2c^5 + 42$

Polynomial

$$3a^2bd - 6b^2c^5 + 42$$

Polynomial representation

Example polynomial: $3a^2bd - 6b^2c^5 + 42$

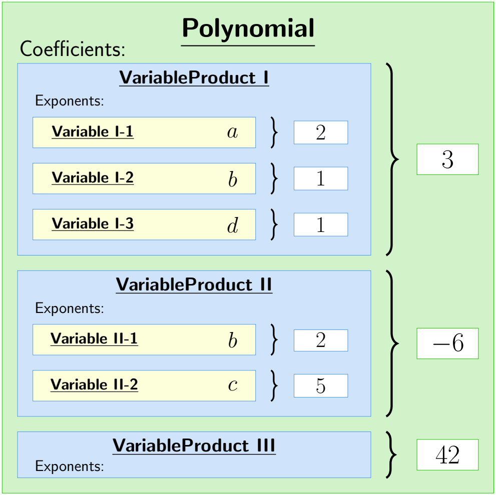
Polynomial

Coefficients:

VariableProduct I a^2bd	3
VariableProduct II b^2c^5	-6
VariableProduct III	42

Polynomial representation

Example polynomial: $3a^2bd - 6b^2c^5 + 42$



Ordering inside a polynomial

Unique ordering of summands allows for efficient polynomial comparison.

- Ordering of variables: $a < b < c < ... < x < y < z$

Ordering inside a polynomial

Unique ordering of summands allows for efficient polynomial comparison.

- Ordering of variables: $a < b < c < ... < x < y < z$
- Ordering of variable products:
  - By exponent of a primarily (higher exponent first)
  - By exponent of b secondarily
  - ...
  - By exponents of z lastly
  - $a^3 < a^2b^2 < a^2b < a^2 < b^2 < b <$ (None)

Ordering inside a polynomial

Unique ordering of summands allows for efficient polynomial comparison.

- Ordering of variables: $a < b < c < ... < x < y < z$
- Ordering of variable products:
  - By exponent of a primarily (higher exponent first)
  - By exponent of b secondarily
  - ...
  - By exponents of z lastly
  - $a^3 < a^2b^2 < a^2b < a^2 < b^2 < b <$ (None)
- Establishes a unique order for sub-terms of the polynomial

Ordering inside a polynomial

Unique ordering of summands allows for efficient polynomial comparison.

- Ordering of variables: $a < b < c < ... < x < y < z$
- Ordering of variable products:
  - By exponent of a primarily (higher exponent first)
  - By exponent of b secondarily
  - ...
  - By exponents of z lastly
  - $a^3 < a^2b^2 < a^2b < a^2 < b^2 < b <$ (None)
- Establishes a unique order for sub-terms of the polynomial
- Equivalent terms are represented as identical polynomials

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse(`"a + b * c"`)

Term strings are `blue`, polynomial objects are $green$.

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse(`"a + b * c"`) | Split off right sub-term "c"
Parse(`"a + b"`) * Parse(`"c"`)

Term strings are `blue`, polynomial objects are *green*.

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse(`"a + b * c"`)	Split off right sub-term "c"
Parse(`"a + b"`) * Parse(`"c"`)	Parse variable "c"
Parse(`"a + b"`) * c	

Term strings are `blue`, polynomial objects are $green$.

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse("a + b * c")	Split off right sub-term "c"
Parse("a + b") * Parse("c")	Parse variable "c"
Parse("a + b") * c	Split off right sub-term "b"
(Parse("a") + Parse("b")) * c	

Term strings are `blue`, polynomial objects are $green$.

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse(`"a + b * c"`)	Split off right sub-term "c"
Parse(`"a + b"`) * Parse(`"c"`)	Parse variable "c"
Parse(`"a + b"`) * c	Split off right sub-term "b"
(Parse(`"a"`) + Parse(`"b"`)) * c	Parse variable "b"
(Parse(`"a"`) + b) * c	

Term strings are `blue`, polynomial objects are $green$.

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse(`"a + b * c"`)	Split off right sub-term "c"
Parse(`"a + b"`) * Parse(`"c"`)	Parse variable "c"
Parse(`"a + b"`) * c	Split off right sub-term "b"
(Parse(`"a"`) $+$ Parse(`"b"`)) * c	Parse variable "b"
(Parse(`"a"`) $+$ b) * c	Parse variable "a"
$(a + b)$ * c	

Term strings are `blue`, polynomial objects are $green$.

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse(`"a + b * c"`)	Split off right sub-term "c"
Parse(`"a + b"`) * Parse(`"c"`)	Parse variable "c"
Parse(`"a + b"`) * c	Split off right sub-term "b"
(Parse(`"a"`) $+$ Parse(`"b"`)) * c	Parse variable "b"
(Parse(`"a"`) $+$ b) * c	Parse variable "a"
(a $+$ b) * c	Add polynomials a and b
$(a+b)$ * c	

Term strings are `blue`, polynomial objects are *green*.

Term parsing: Left-to-right? Right-to-left!

Terms should be evaluated left-to-right: `a + b * c` $\equiv$ `(a + b) * c`
Parse sub-terms right-to-left!

Parse(`"a + b * c"`)	Split off right sub-term "c"
Parse(`"a + b"`) * Parse(`"c"`)	Parse variable "c"
Parse(`"a + b"`) * c	Split off right sub-term "b"
(Parse(`"a"`) + Parse(`"b"`)) * c	Parse variable "b"
(Parse(`"a"`) + b) * c	Parse variable "a"
(a + b) * c	Add polynomials a and b
$(a + b)$ * c	Multiply polynomials $a + b$ and c
$ac + bc$	

Term strings are `blue`, polynomial objects are *green*.

How to handle brackets and integers

Brackets can be handled the same way: Split the entire bracket off and perform a recursive call to parse that sub-term first. Integers are treated like variables.

Parse(`"a + (17 * c)"`)

Term strings are `blue`, polynomial objects are $green$.

How to handle brackets and integers

Brackets can be handled the same way: Split the entire bracket off and perform a recursive call to parse that sub-term first. Integers are treated like variables.

Parse(`"a + (17 * c)"`) | Split off right sub-term "(17 ∗ c)"
Parse(`"a"`) + Parse(`"17 * c"`)

Term strings are `blue`, polynomial objects are *green*.

How to handle brackets and integers

Brackets can be handled the same way: Split the entire bracket off and perform a recursive call to parse that sub-term first. Integers are treated like variables.

Parse(`"a + (17 * c)"`)	Split off right sub-term "(17 ∗ c)"
Parse(`"a"`) + Parse(`"17 * c"`)	Split off right sub-term "c"
Parse(`"a"`) + (Parse(`"17"`) * Parse(`"c"`))	Parse variable "c"
Parse(`"a"`) + (Parse(`"17"`) * c)	Parse integer "17"
Parse(`"a"`) + (17 * c)	Multiply polynomials 17 and c
Parse(`"a"`) + $17c$	Parse variable "a"
a + $17c$	Add polynomials a and $17c$
$a + 17c$	

Term strings are `blue`, polynomial objects are *green*.

Parsing algorithm: Alternatives

I used a recursive algorithm to parse the terms into polynomials.
Other ideas?

- Iterative instead of recursive
  - Go through the term left-to-right
  - Harder to handle brackets: Would need to store parsed sub-terms in a more complex data structure
- Don't parse into polynomials at all
  - Substitute concrete values for each variable and evaluate
    $\Rightarrow$ Very hard to be *sure* that the terms are equivalent
- Other approaches possible

Implementation in C#

Representing polynomials and variable products as objects:

- Custom addition, subtraction, multiplication operators
- Custom equality checks
  $\Rightarrow$ Efficient use of built-in types such as dictionaries
- Custom ordering for variable products
  $\Rightarrow$ Efficient use of *sorted* dictionaries

Allows for readable code like this:

C# example on how to use the classes

```
Polynomial term1 = Polynomial.Parse("a + b");
Polynomial term2 = Polynomial.Parse("2*a");
Polynomial diff = term1 - term2;            // diff is -a + b

Polynomial term3 = Polynomial.Parse("b - a");

// diff and term3 are both -a + b and are treated as identical
Debug.Assert(diff.Equals(term3));
```

Any questions?

Integers
- ▷ non-negative
- ▷ (single-digit)

Variables
- ▷ a to z

`(3*a) - c + (b*b)*8`

Brackets
- ▷ matching
- ▷ can be nested

Operators
- ▷ +, - and *
- ▷ equal precedence