

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis .....</b>	<b>1</b>
<b>Beispiel 1 Arithmetische Ausdrücke (infix) .....</b>	<b>2</b>
<b>Lösungsidee .....</b>	<b>2</b>
Grammatik.....	2
Scanner .....	3
Parser .....	4
<b>Testfälle .....</b>	<b>6</b>
void test_1();.....	6
void test_2();.....	6
void test_3();.....	6
void test_4();.....	7
void test_5();.....	7
<b>Beispiel 2 Arithmetische Ausdrücke (präfix) .....</b>	<b>8</b>
<b>Lösungsidee .....</b>	<b>8</b>
Grammatik.....	8
Scanner .....	8
Parser .....	9
<b>Testfälle .....</b>	<b>11</b>
void test_1();.....	11
void test_2();.....	11
void test_3();.....	11
void test_4();.....	11
void test_5();.....	12
<b>Beispiel 3 Rechnen mit Variablen .....</b>	<b>13</b>
<b>Lösungsidee .....</b>	<b>13</b>
<b>Testfälle .....</b>	<b>14</b>
void test_1();.....	14
void test_2();.....	14
void test_3();.....	14
void test_4();.....	15
void test_5();.....	15

# Beispiel 1 Arithmetische Ausdrücke (infix)

## Lösungsidee

Es soll ein Programm zur Auswertung von arithmetischen Ausdrücken in Infix-Notation implementiert werden. Hierfür soll ein Parser Modul erstellt werden, welches mit Hilfe des zu Verfügung gestellten pro-facilities/scanner und der vergebenen Grammatik Arithmetische Ausdrücke überprüfen und diese anwenden können soll.

Um etwaige Fehler behandeln zu können sollen Exceptiones für das Programm erstellt werden.

## Grammatik

Die Grundelemente aus denen alle anderen Teile der Grammatik bestehen sind Digit, MultOp und AddOp. Ein Digit besteht aus einer Zahlen von 0-9. Der AddOp beinhaltet + und -. Der MultOp beinhaltet dann / und \*.

Der nächstkleinere Ausdruck ist der Unsigned, dieser besteht mindestens aus einem Digit, kann aber auch mehrere umfassen. Von der Größenordnung her kommt als nächstes der Factor. Dieser hat optional einen AddOp, also ein positives oder negatives Vorzeichen, gefolgt von einem Unsigned oder einer PExpression.

Eine PExpression besteht aus einer öffnenden Klammer, einer Expression und einer schließenden Klammer. Eine Expression besteht aus mindestens einem Term und einem AddOp gefolgt von noch einem Term. Der Ausdruck Term wiederum besteht aus mindestens einem Factor gefolgt von einem MultOp und einem weiteren Factor, falls vorhanden.

$T_s = \{ '+', '-', '/', '*' \}$  – Terminal Symbole

$T_k = \{ \text{Real, Identifier} \}$  - Terminal Klassen

Non-Terminal = { Ausdruck, ... }

```

Expression = Term { AddOp Term } .
Term       = Factor { MultOp Factor } .
Factor     = [ AddOp ] ( Unsigned | PExpression ) .
Unsigned   = Digit { Digit } .
PExpression = „ ( “ Expression „ ) “ .
AddOp      = „+“ | „-“ .
MultOp     = „.“ | „/“ .
Digit      = „0“ | „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“ .

```

## 1 Arithmetische Ausdrücke Grammatik

### Scanner

Der verwendete Scanner verwendet als Datentyp zum Einlesen von Zeichen char. Es wird C++20 für dessen Verwendung vorausgesetzt. Der Scanner hat viele Methoden mit denen er die Daten, die ihm der Parser übergibt, einlesen kann. Siehe Anwender Doku.

Der Scanner erkennt die folgenden Terminalsymbole:

Symbol	Symbol	Symbol
= assign	^ caret	: colon
, comma	/ division	" double quote
( left parenthesis	- minus	* multiply
. period	+ plus	) right parenthesis
; semicolon	␣ space	\t tabulator
_ underscore	\n end of line (eol)	

Darüber hinaus wird auch *end of file* (eof) als Terminalsymbol interpretiert.

## 2 Vorlesungsunterlagen

Der Scanner erkennt die folgenden Terminalklassen:

Symbol	Attributabfrage	Datentyp
<i>identifizier</i>	get_identifizier()	std::string
<i>integer</i>	get_integer() oder get_number()	int oder double
<i>keyword</i>		
<i>real</i>	get_real() oder get_number()	double
<i>string</i>	get_string()	std::string

## 3 Vorlesungsunterlagen

Arbeitsaufwand: 12 h

Der Scanner verwendet intern die folgenden Grammatiken:

```
Identifier = Alpha { Alpha | Digit } .  
Integer    = Digit { Digit } .  
Real      = ( Integer „.“ [ Integer ] ) | ( „.“ Integer ) .  
String     = „“ { any char except quote, eol, or eof } „“ .
```

```
Alpha = „a“ | ... | „z“ | „A“ | ... | „Z“ | „-“ .  
Digit = „0“ | „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“ .
```

```
BlockComment = „/*“ { any char except eof } „*/“ .  
LineComment  = „//“ { any char except eol or eof } „eol“ .
```

#### 4 Vorlesungsunterlagen

Auch der Scanner verwendet das Digit, Zahlen von 0-9, und etwas, dass hier Alpha heißt und das chars beinhaltet. Der Ausdruck Integer besteht aus mindestens einem oder mehreren Digits. Ein Identifier ist mindestens ein Alpha gefolgt von optional einem weiteren Alpha oder einem Digit.

Ein String beginnt mit „, umfasst dann alles an chars was der Compiler auswerten kann und endet mit „. Ein Real besteht aus einem Integer.Integer oder einem .Integer.

## Parser

Der Parser für diese Modul beinhaltet die bei Grammatik erklärte Grammatik. Eine allgemeine Exception und eine speziellere div\_by\_zero. Der Parser greift aus den Scanner zu.

```
//exceptions  
class ParseException final : public runtime_error {  
public:  
    explicit ParseException(string const& message) : runtime_error{ message } { }  
};  
  
class div_by_zero_error : public exception {  
public:  
    virtual char const* what() const throw() {  
        return "Error div by zero!";  
    }  
};
```

```
//parser
class ArithmeticParser {
public:
    double parse(istream& in);
    double parse(string const& filename);

private:
    //predicates
    bool is_tb_Expression() const;
    bool is_tb_Term() const;
    bool is_tb_Factor() const;
    bool is_tb_AddOp() const;
    bool is_tb_MultOp() const;
    bool is_tb_PExpression() const;
    bool is_tb_Unsigned() const;

    double parse_Expression();
    double parse_Term();
    double parse_Factor();
    double parse_AddOp();
    double parse_PExpression();

    pfc::scanner m_scanner;
};
```

Die is\_tb\_ Funktionen überprüfen, ob ein Objekt die Grammatik Regeln für den jeweiligen Ausdruck erfüllt. Die beiden öffentlichen parse Funktionen sind dafür da Dateien, bzw. Dinge von einem Strom einzulesen und aufeinander weiterzuleiten, bzw. auf die jeweiligen parse\_ Funktionen weiterzuleiten, bis zum jeweiligen Rekursionsboden. M\_scanner legt einfach einen Scanner für die Klasse an, welcher dann benutzt werden kann.

## Testfälle

**void test\_1();**

Einzelne Werte – positiv & negativ. User Eingabe über die Konsole.



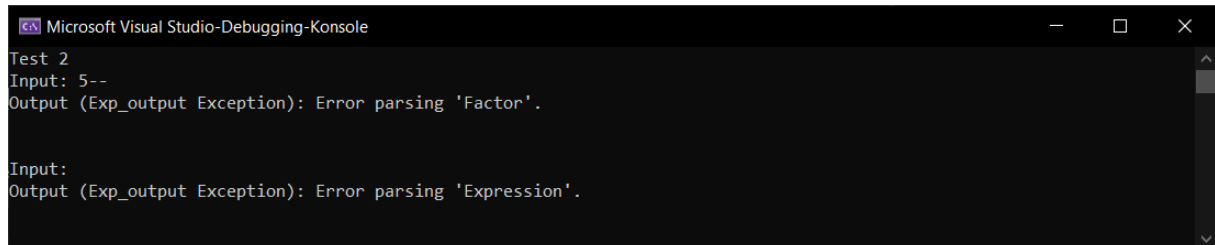
```
Microsoft Visual Studio-Debugging-Konsole
Test 1
Input: 4
Output (Exp_output 4): 4

Input: -4
Output (Exp_output -4): -4

0
Input: 0
Output (Exp_output ?): 0
```

**void test\_2();**

Leere Eingabe und falsche Eingabe.



```
Microsoft Visual Studio-Debugging-Konsole
Test 2
Input: 5--
Output (Exp_output Exception): Error parsing 'Factor'.

Input:
Output (Exp_output Exception): Error parsing 'Expression'.
```

**void test\_3();**

Durch Null dividieren.



```
Microsoft Visual Studio-Debugging-Konsole
Test 3
Input: 5/0
Output (Exp_output Exception): Error div by zero!
```

Vanessa Wahlmüller

`void test_4();`

Alle vier Rechenoperatoren +, -, /, \*



```
Microsoft Visual Studio-Debugging-Konsole

Test 4
Input: 5+5
Output (Exp_output 10): 10

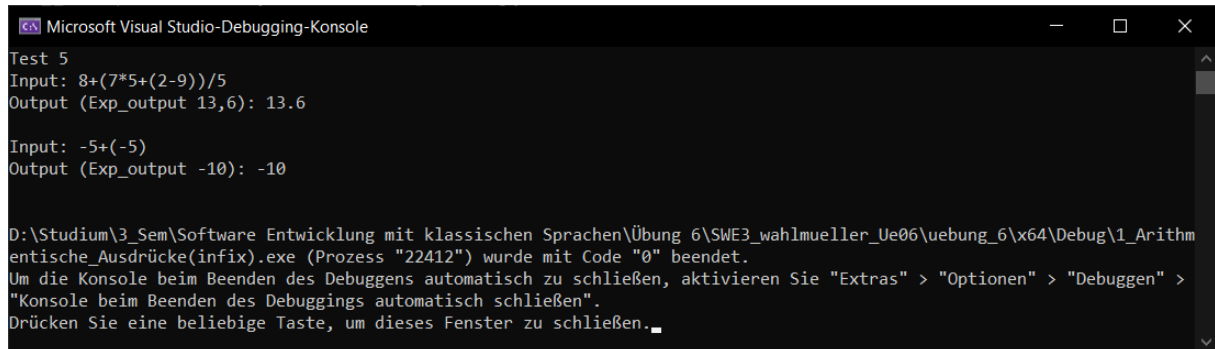
Input: 3-8
Output (Exp_output -5): -5

Input: 3*5
Output (Exp_output 15): 15

Input: 5/5
Output (Exp_output 1): 1
```

`void test_5();`

Längere Rechnung mit Klammern und negative Zahlen addieren.



```
Microsoft Visual Studio-Debugging-Konsole

Test 5
Input: 8+(7*5+(2-9))/5
Output (Exp_output 13,6): 13.6

Input: -5+(-5)
Output (Exp_output -10): -10

D:\Studium\3_Sem\Software Entwicklung mit klassischen Sprachen\Übung 6\SWE3_wahlmueller_Ue06\uebung_6\x64\Debug\1_Arithm
entische_Ausdrücke(infix).exe (Prozess "22412") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen, aktivieren Sie "Extras" > "Optionen" > "Debuggen" >
"Konsole beim Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Arbeitsaufwand: 12 h

## Beispiel 2 Arithmetische Ausdrücke (präfix)

### Lösungsidee

Wie auch schon in der Aufgabe zuvor soll ein Parser implementiert werden, nur soll dieser in Grammatik präfix Schreibweise erhalten und damit rechnen können.

#### Grammatik

Expression = Term|Digit

Term = Operation „space“ Term|Digit „space“ Term|Digit

Operator = „+“ | „-“ | „:“ | „\*“

Digit = „0“ | „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“

Die einfachsten Ausdrücke sind Operator und Digit, diese beinhalten einerseits die Zahlen von 0-9 (Digit) und andererseits +, -, \*, / (Operator). Der nächstkomplexere Ausdruck ist die Expression. Diese kann aus einem Digit oder aus einem Term bestehen. Ein Term besteht aus einem Operator gefolgt von einem Leerzeichen, gefolgt von einem Term oder Digit.

#### Scanner

Der verwendete Scanner verwendet als Datentyp zum Einlesen von Zeichen char. Es wird C++20 für dessen Verwendung vorausgesetzt. Der Scanner hat viele Methoden mit denen er die Daten, die ihm der Parser übergibt, einlesen kann. Siehe Anwender Doku.

Der Scanner erkennt die folgenden Terminalsymbole:

Symbol	Symbol	Symbol
= assign	^ caret	: colon
, comma	/ division	" double quote
( left parenthesis	- minus	* multiply
. period	+ plus	) right parenthesis
; semicolon	␣ space	\t tabulator
_ underscore	\n end of line (eol)	

Darüber hinaus wird auch *end of file* (eof) als Terminalsymbol interpretiert.



Der Scanner erkennt die folgenden Terminalklassen:

Symbol	Attributabfrage	Datentyp
<i>identifier</i>	<code>get_identifier()</code>	<code>std::string</code>
<i>integer</i>	<code>get_integer()</code> oder <code>get_number()</code>	<code>int</code> oder <code>double</code>
<i>keyword</i>		
<i>real</i>	<code>get_real()</code> oder <code>get_number()</code>	<code>double</code>
<i>string</i>	<code>get_string()</code>	<code>std::string</code>

#### 6 Vorlesungsunterlagen

Der Scanner verwendet intern die folgenden Grammatiken:

```
Identifier = Alpha { Alpha | Digit } .  
Integer   = Digit { Digit } .  
Real      = ( Integer „ . “ [ Integer ] ) | ( „ . “ Integer ) .  
String    = „ “ { any char except quote, eol, or eof } „ “ .
```

```
Alpha = „ a “ | ... | „ z “ | „ A “ | ... | „ Z “ | „ - “ .  
Digit = „ 0 “ | „ 1 “ | „ 2 “ | „ 3 “ | „ 4 “ | „ 5 “ | „ 6 “ | „ 7 “ | „ 8 “ | „ 9 “ .
```

```
BlockComment = „ / * “ { any char except eof } „ * / “ .  
LineComment  = „ / / “ { any char except eol or eof } „ eol “ .
```

#### 7 Vorlesungsunterlagen

Auch der Scanner verwendet das Digit, Zahlen von 0-9, und etwas, dass hier Alpha heißt und das chars beinhaltet. Der Ausdruck Integer besteht aus mindestens einem oder mehreren Digits. Ein Identifier ist mindestens ein Alpha gefolgt von optional einem weiteren Alpha oder einem Digit.

Ein String beginnt mit „, umfasst dann alles an chars was der Compiler auswerten kann und endet mit „. Ein Real besteht aus einem Integer.Integer oder einem .Integer.

## Parser

Der Parser für diese Modul beinhaltet die bei Grammatik erklärte Grammatik. Eine allgemeine Exception und eine speziellere `div_by_zero`. Der Parser greift aus den Scanner zu.

```
//exceptions
class ParseException final : public runtime_error {
public:
    explicit ParseException(string const& message) : runtime_error{ message } { }
};

class div_by_zero_error :public exception {
public:
    virtual char const* what() const throw() {
        return "Error div by zero!";
    }
};
```

```
32 class ArithmeticParser {
33 public:
34     double parse(istream& in);
35     double parse(string const& filename);
36
37 private:
38     //predicates
39     bool is_tb_Expression() const; //= Term|Digit
40     bool is_tb_Term() const; //= Operation „space“ Term|Digit
41     bool is_tb_Op() const; //= „+“ | „-“ | „:“ | „*“
42     bool is_tb_AddOp() const; //= „+“ | „-“
43     bool is_tb_MultOp() const; //= „:“ | „/“
44     bool is_tb_Unsigned() const; // - Digit = „0“ | „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“
45
46     double parse_Expression();
47     double parse_Term();
48     double parse_MultOp();
49     double parse_AddOp();
50
51     pfc::scanner m_scanner;
52
53 };
```

Die is\_tb\_ Funktionen überprüfen, ob ein Objekt die Grammatik Regeln für den jeweiligen Ausdruck erfüllt. Die beiden öffentlichen parse Funktionen sind dafür da Dateien, bzw. Dinge von einem Strom einzulesen und aufeinander weiterzuleiten, bzw. auf die jeweiligen parse\_ Funktionen weiterzuleiten, bis zum jeweiligen Rekursionsboden. M\_scanner legt einfach einen Scanner für die Klasse an, welcher dann benutzt werden kann.

Vanessa Wahlmüller

## Testfälle

**void test\_1();**

Einzelne Zahlen positive und negativ. Eingabe des Users.

```
Microsoft Visual Studio-Debugging-Konsole
Test 1
Input: + 4
Output (Exp_output 4): 4

Input: - 0 4
Output (Exp_output -4): -4

0
Input: 0
Output (Exp_output ?): 0
```

**void test\_2();**

Leere Eingabe und blödsinnige Eingabe.

```
Microsoft Visual Studio-Debugging-Konsole
Test 2
Input: ()
Output (Exp_output Exception): Error parsing 'Expression'.

Input:
Output (Exp_output Exception): Error parsing 'Expression'.
```

**void test\_3();**

Divison durch null.

```
Microsoft Visual Studio-Debugging-Konsole
Test 3
Input: / 5 0
Output (Exp_output Exception): Error div by zero!
```

**void test\_4();**

Alle Operatoren die implementiert wurden. +, -, /, \*

```
Auswählen Microsoft Visual Studio-Debugging-Konsole
Test 4
Input: + 5 5
Output (Exp_output 10): 10

Input: - 3 8
Output (Exp_output -5): -5

Input: * 3 5
Output (Exp_output 15): 15

Input: / 5 5
Output (Exp_output 1): 1
```

Arbeitsaufwand: 12 h

Vanessa Wahlmüller

`void test_5();`

Längere Rechnungen in der präfix Notation



A screenshot of a Microsoft Visual Studio Debugging Console window. The title bar reads "Auswählen Microsoft Visual Studio-Debugging-Konsole". The console output shows two test cases. The first test case is labeled "Test 5" and shows an input expression `/ + 4 9 + 17 2` and an output `Output (Exp_output 0,684211): 0.684211`. The second test case shows an input expression `+ - 0 5 - 0 5` and an output `Output (Exp_output -10): -10`. The console has a dark background and a scrollbar on the right side.

```
Auswählen Microsoft Visual Studio-Debugging-Konsole

Test 5
Input: / + 4 9 + 17 2
Output (Exp_output 0,684211): 0.684211

Input: + - 0 5 - 0 5
Output (Exp_output -10): -10
```

Arbeitsaufwand: 12 h

## Beispiel 3 Rechnen mit Variablen

### Lösungsidee

Siehe Lösungsidee von Beispiel 1 für Parser, Scanner und Grammatik.

Es sollen auch variablen für Rechnungen möglich sein. Diese müssen zuvor festgelegt werden können. Wird versucht mit variablen zu rechnen, die nicht festgelegt wurden, so soll eine entsprechende Fehlermeldung ausgegeben werden.

Der Ausdruck des Digits wird nun um vordefinierte variablen erweitert und die bereits implementierten parse\_ Funktionen werden leicht verändert und is\_tb\_ wurde um is\_tb\_Variable erweitert. Ebenfalls wurde eine eigene Exception hinzugefügt.

```
class variable_exception :public exception {
public:
    virtual char const* what() const throw() {
        return "Error variables!";
    }
};
```

```
class ArithmeticParser {
public:
    double parse(istream& in);
    double parse(string const& filename);
    void set_variables(string const& variable, int const number); //to define variables

private:
    bool is_tb_Expression() const;
    bool is_tb_Term() const;
    bool is_tb_Factor() const;
    bool is_tb_AddOp() const;
    bool is_tb_MultOp() const;
    bool is_tb_PExpression() const;
    bool is_tb_Unsigned() const;
    bool is_tb_Variable() const; //check if variable

    double parse_Expression();
    double parse_Term();
    double parse_Factor();
    double parse_AddOp();
    double parse_PExpression();

    pfc::scanner m_scanner;
    map<string, int> variables; //for storing variables
};
```

## Testfälle

### `void test_1();`

Einzelene Variablen mit einfachen Zahlen belegt – negativ und positiv. User eingabe – x, y, z wurden für den User hinterlegt.



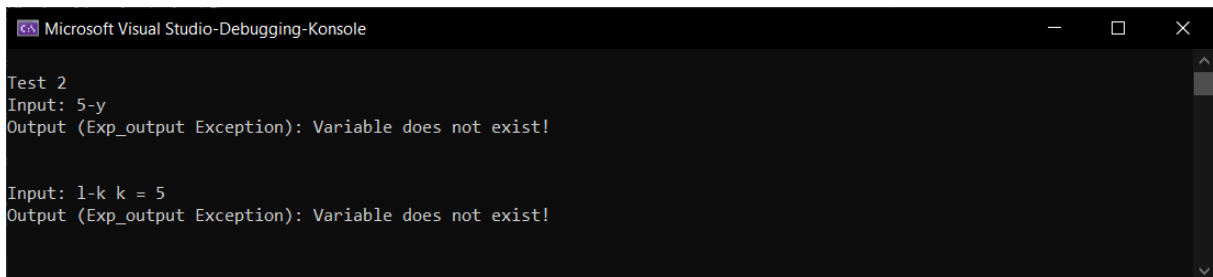
```
Microsoft Visual Studio-Debugging-Konsole
Test 1
Input: x = 4
Output (Exp_output 4): 4

Input: y = -4
Output (Exp_output -4): -4

x+z*y
Input: x+z*y x = 12, y = 4, z = -8
Output (Exp_output ?): -20
```

### `void test_2();`

Rechnung mit einer Zahl und einer nicht belegten Variable und zwei Variablen, eine davon nicht belegt.

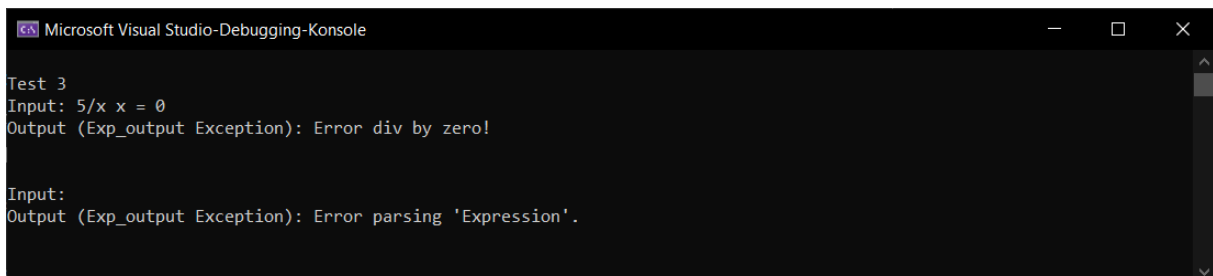


```
Microsoft Visual Studio-Debugging-Konsole
Test 2
Input: 5-y
Output (Exp_output Exception): Variable does not exist!

Input: 1-k k = 5
Output (Exp_output Exception): Variable does not exist!
```

### `void test_3();`

Division durch Null und leerer Ausdruck.



```
Microsoft Visual Studio-Debugging-Konsole
Test 3
Input: 5/x x = 0
Output (Exp_output Exception): Error div by zero!

Input:
Output (Exp_output Exception): Error parsing 'Expression'.
```

Vanessa Wahlmüller

`void test_4();`

Alle vier Operatoren mit variablen. +, -, /, \*



```
Microsoft Visual Studio-Debugging-Konsole

Test 4
Input: 5+x x = 5
Output (Exp_output 10): 10

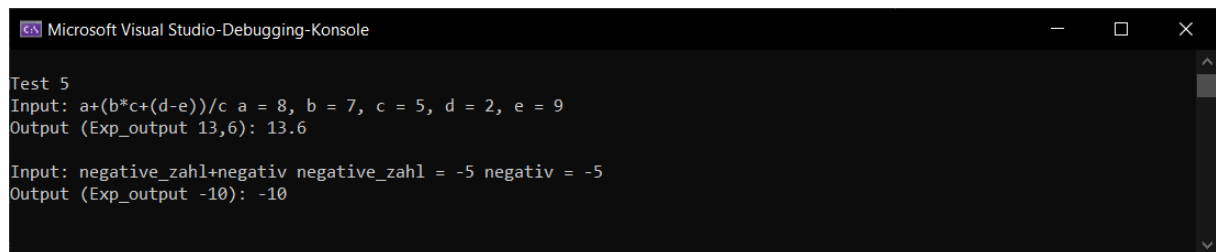
Input: y-8 y = 3
Output (Exp_output -5): -5

Input: 3*z z = 5
Output (Exp_output 15): 15

Input: 5/a a = 5
Output (Exp_output 1): 1
```

`void test_5();`

Lange Rechnung mit vielen Variablen und Rechnung mit negativen Variablen.



```
Microsoft Visual Studio-Debugging-Konsole

Test 5
Input: a+(b*c+(d-e))/c a = 8, b = 7, c = 5, d = 2, e = 9
Output (Exp_output 13,6): 13.6

Input: negative_zahl+negativ negative_zahl = -5 negativ = -5
Output (Exp_output -10): -10
```

Arbeitsaufwand: 12 h