

Name: Sandra Straube **Aufwand in h:** 3

Punkte: **Kurzzeichen Tutor/in:**

Beispiel 1 (30 Punkte): Arithmetische Ausdrücke (infix)

Komplettieren Sie das in der Übung begonnene Program zur Auswertung von arithmetischen Ausdrücken in Infix-Notation mit Hilfe des `pro-facilities/scanner`. Behandeln Sie Fehler wie z. B. *division by zero* mit Exceptions. Testen Sie ausführlich.

Beispiel 2 (30 Punkte): Arithmetische Ausdrücke (präfix)

Entwerfen Sie eine Grammatik zur Berechnung von arithmetischen Ausdrücken in Präfix-Notation. Notieren Sie diese Grammatik in EBNF-Schreibweise und implementieren Sie sie mit Hilfe des `pro-facilities/scanner`. Behandeln Sie Fehler wie z. B. *division by zero* mit Exceptions. Testen Sie ausführlich.

Beispiel 3 (40 Punkte): Rechnen mit Variablen

Erweitern Sie Ihr Programm von Beispiel 1 so, dass im Input nicht nur Literale sondern auch zuvor definierte Variablen vorkommen dürfen. Verwenden Sie z. B. eine `std::map<>`, um Variablennamen auf Werte abzubilden. Enthält ein Ausdruck undefinierte Variablen, so wird entsprechend reagiert. Testen Sie ausführlich.

Anmerkungen: (1) Geben Sie für Ihre Problemlösungen auch Lösungsideen an. (2) Kommentieren Sie Ihre Algorithmen ausführlich. (3) Strukturieren Sie Ihre Programme sauber. (4) Geben Sie ausreichend Testfälle ab und prüfen Sie alle Eingabedaten auf ihre Gültigkeit.

[Inhaltsverzeichnis](#)

Beispiel 1 Arithmetische Ausdrücke (infix):	1
Lösungsidee.....	1
Testfälle	3
Fall 1	3
Fall 2.....	3
Fall 3.....	3
Fall 4.....	4
Fall 5.....	4
Fall 6.....	4
Fall 7.....	5
Beispiel 2 Arithmetische Ausdrücke (präfix):	6
Lösungsidee.....	6
Testfälle	7
Fall 1	7
Fall 2.....	7
Fall 3.....	7
Fall 4.....	8
Fall 5.....	8
Fall 6.....	8
Beispiel 3 Rechnen mit Variablen:	9
Lösungsidee.....	9
Testfälle	10
Fall 1	10
Fall 2.....	10

Fall 3.....	10
Fall 4.....	10
Fall 5.....	11
Fall 6.....	11
Fall 7.....	12
Fall 8.....	12

Beispiel 1 Arithmetische Ausdrücke (infix):

Lösungsidee

Es soll das in der Übung angefangene Scanner & Parser-Beispiel vervollständigt werden, wozu die Unterlagen aus der Vorlesung verwendet werden sollen. Es wird C++ 20 vorausgesetzt.

Der aus der Vorlesung gegebene Scanner durchsucht einen Ausdruck, ohne ihn zu parsen oder zu prüfen. Mithilfe des Scanners sollen Terminalsymbole und Terminalklassen erkannt, geprüft und zurückgegeben werden können. Hierzu werden Nicht-Terminal-Symbole soweit durch Produktionsregeln aufgespalten, bis Terminalsymbole erhalten werden. Durch die Terminalklassen sollen z.B. Nummern oder Strings herausgefiltert/verwendet werden. Der Scanner verwendet hierbei folgende Grammatiken:

```
1  Identifier = Alpha { Alpha | Digit } .
2  Integer = Digit { Digit } .
3  Real = (Integer "." [ Integer ] )|("." Integer).
4  String = "" { any char except quote, eol, or eof } "" .
5
6  Alpha = "a" | ... | "z" | "A" | ... | "Z" | "_" .
7  Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
8
9  BlockComment = "/*"{ any char except eof }"*/" .
10 LineComment = "//"{ any char except eol or eof }"eol".
```

Diese Grammatiken sind in der Erweiterten Backus-Naur-Form (EBNF) geschrieben. Dies bezeichnet eine standardisierte Form der Darstellung einer kontextfreien Grammatik, indem Nichtterminal-Symbole in Terminalsymbole aufgesplittet werden.

EBNF kann mithilfe des Parsers entsprechend umgesetzt werden, um einen Arithmetischen Ausdruck in Infix zu berechnen. Hierfür gilt folgende Grammatik:

```
1  Expression = Term { AddOp Term }.
2  Term       = Factor { MultOp Factor }.
3  Factor     = [ AddOp ] ( Number | PExpression ).
4  PExpression = "(" Expression ")" .
5  AddOp      = "+" | "-" .
6  MultOp     = "*" | "/" .
7  Number     = Die Grammatik für Number ist im Scanner eingebaut (Integer | Real).
```

Hierzu wird eine Expression zunächst in einen Term reduziert, wobei kein oder beliebig viele weitere Terme in Kombination mit einem AddOp folgen können. Ein Term ist hierbei ein Factor, welcher wiederum auf keinem oder mehreren Factor mit einem MultOp vorkommen können. Ein Factor ist genau ein oder kein AddOp und darauf folgend eine Nummer oder eine Expression in Klammern. Number wird hier automatisch bereits im Scanner korrekt herausgelesen. Mithilfe dieser Grammatik soll ein Parser nun feststellen können ob folgender Ausdruck in Infix korrekt ist und diesen dann entsprechend umsetzen:

$((17 * -4) + 8) / -6.$

Mithilfe diverser bool-Methoden sollen übergebene Werte vom Scanner geprüft werden, ob sie der Grammatik an der aktuellen Stelle entsprechen. Wenn dem so ist, soll die entsprechende Grammatik geparsed werden. Im Endeffekt soll der gesamte Ausdruck umgesetzt werden.

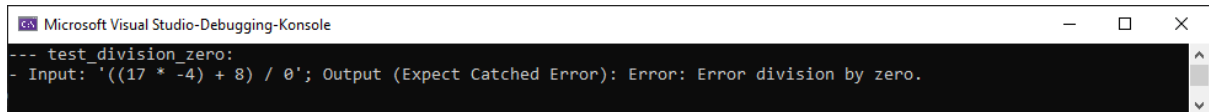
Es soll darauf geachtet werden, dass Fehler wie Division durch 0 abgefangen werden und alles korrekt funktioniert, auch geschachtelte Klammern, negative Werte, sämtliche Operationen. Auch sollen leere Werte, falsche Zeichen und falsche Grammatik gehandhabt werden.

Testfälle

Fall 1: Test erfolgreich

Teste Division durch 0.

Output: **bool** test_division_zero()

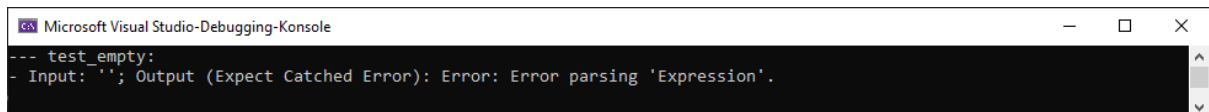


```
Microsoft Visual Studio-Debugging-Konsole
--- test_division_zero:
- Input: '((17 * -4) + 8) / 0'; Output (Expect Caught Error): Error: Error division by zero.
```

Fall 2: Test erfolgreich

Teste leeren Ausdruck.

Output: **bool** test_empty()



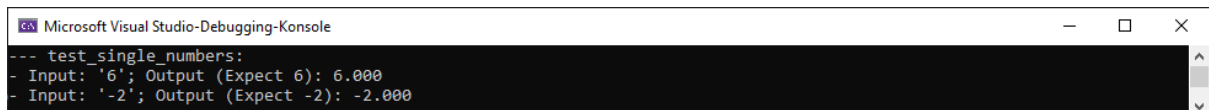
```
Microsoft Visual Studio-Debugging-Konsole
--- test_empty:
- Input: ''; Output (Expect Caught Error): Error: Error parsing 'Expression'.
```

Fall 3: Test erfolgreich

Teste

- einzelne, positive Zahl
- einzelne, negative Zahl.

Output: **bool** test_single_numbers()

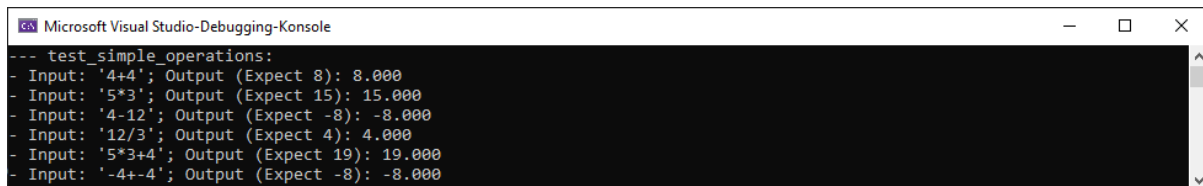


```
Microsoft Visual Studio-Debugging-Konsole
--- test_single_numbers:
- Input: '6'; Output (Expect 6): 6.000
- Input: '-2'; Output (Expect -2): -2.000
```

Fall 4: Test erfolgreich

Teste diverse Rechenoperationen:

- Add
- Mult
- Sub
- Div
- Mehrere Operationen
- Addition zweier negativer Zahlen.

Output: `bool test_simple_operations()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_simple_operations:
- Input: '4+4'; Output (Expect 8): 8.000
- Input: '5*3'; Output (Expect 15): 15.000
- Input: '4-12'; Output (Expect -8): -8.000
- Input: '12/3'; Output (Expect 4): 4.000
- Input: '5*3+4'; Output (Expect 19): 19.000
- Input: '-4+-4'; Output (Expect -8): -8.000
```

Fall 5: Test erfolgreich

Teste einfachen Ausdruck mit Klammern.

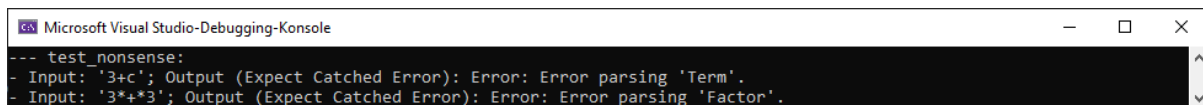
Output: `bool test_parenthesis()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_parenthesis:
- Input: '3*(4+5)'; Output (Expect 27): 27.000
```

Fall 6: Test erfolgreich

Teste auf

- Buchstaben (keine Variablen)
- Mehrere Operationen ohne Zusammenhang.

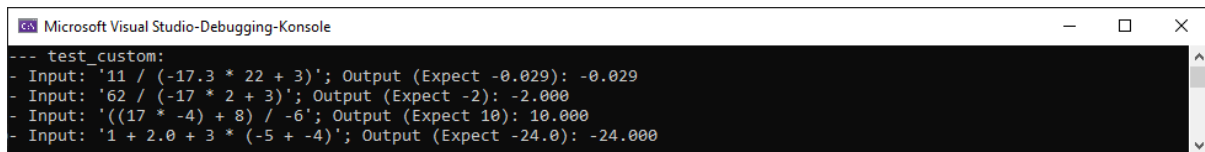
Output: `bool test_nonsense()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_nonsense:
- Input: '3+c'; Output (Expect Caught Error): Error: Error parsing 'Term'.
- Input: '3*+*3'; Output (Expect Caught Error): Error: Error parsing 'Factor'.
```

Fall 7: Test erfolgreich

Teste auf

- Negativer Double in Klammern
- Kein double und negativer Wert in Klammern
- Division durch negativen Wert und geschachtelte Klammern
- Multiplikation vor geklammerten, negativen Ausdruck.

Output: `bool test_custom()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_custom:
- Input: '11 / (-17.3 * 22 + 3)'; Output (Expect -0.029): -0.029
- Input: '62 / (-17 * 2 + 3)'; Output (Expect -2): -2.000
- Input: '((17 * -4) + 8) / -6'; Output (Expect 10): 10.000
- Input: '1 + 2.0 + 3 * (-5 + -4)'; Output (Expect -24.0): -24.000
```


Beispiel 2 Arithmetische Ausdrücke (präfix):

Lösungsidee

Das Beispiel 1 soll hier so umgewandelt werden, dass Präfix- statt Infix-Notationen mithilfe des vorgegebenen Scanners erkannt und mithilfe des Parsers umgesetzt werden können. Da der Aufbau und die Handhabung von Präfix-Notation anders ist, wurde dafür eine neue Grammatik entworfen:

```
1 Expression = ( Term | Number ).
2 Term      = Operator Expression Expression.
3 Operator  = "+" | "-" | ":" | "*".
4 Number    = Die Grammatik für Number ist im Scanner eingebaut (Integer | Real).
```

Eine Expression ist hier entweder eine Nummer oder ein Term. Ein Term besteht wiederum aus einem Operator und nachfolgend zwei Expression, die wiederum ein Term oder eine Nummer sein können. Wichtig hierbei ist, dass negative Zahlen nichtmehr wie in Infix (z.B. -5) angegeben werden können, sondern stattdessen von 0 abgezogen werden müssen (z.B. 0 - 5), wodurch wiederum ein Term. Diese Umstrukturierung wird an dieser Stelle nicht automatisch vom Parser umgesetzt, stattdessen wird dies vorher vorausgesetzt.

Eine Notation in Infix, zum Beispiel $11 / (-17.3 * 22 + 3)$, sieht in Präfix wie folgt aus: $/ 11 + * - 0 17.3 22 3$.

Die Grammatik wurde an diese Struktur angepasst, vorausgesetzt, sämtliche Operatoren sind immer zweistellig. Zu bemerken ist, dass Klammern in Präfix nicht vorhanden, bzw. aufgelöst worden sind. In Präfix wird von links nach rechts gerechnet, da Operatoren vorangestellt sind, weswegen Klammern nicht benötigt werden. Entsprechend sollen Klammeroperationen nicht beachtet werden, auch in Tests.

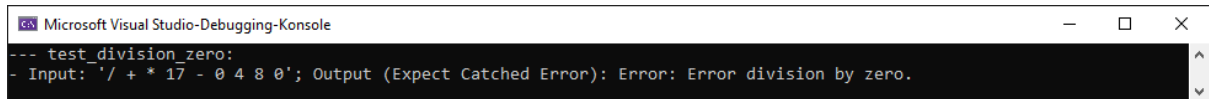
Es sollen sämtliche Ausdrücke von Infix in Präfix funktionieren, ebenfalls soll wieder auf Division durch 0 geachtet werden und auf alles weitere, was in Beispiel 1 berücksichtigt wurde.

Testfälle

Fall 1: Test erfolgreich

Teste Division durch 0.

Output: `bool test_division_zero()`

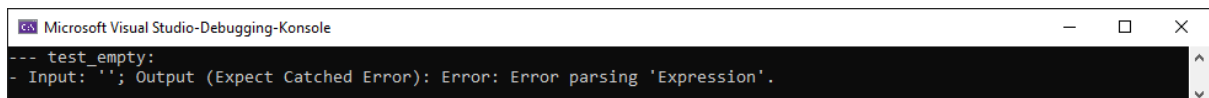


```
Microsoft Visual Studio-Debugging-Konsole
--- test_division_zero:
- Input: '/ + * 17 - 0 4 8 0'; Output (Expect Caught Error): Error: Error division by zero.
```

Fall 2: Test erfolgreich

Teste leeren Ausdruck.

Output: `bool test_empty()`



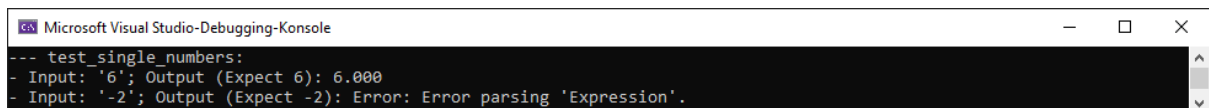
```
Microsoft Visual Studio-Debugging-Konsole
--- test_empty:
- Input: ''; Output (Expect Caught Error): Error: Error parsing 'Expression'.
```

Fall 3: Test erfolgreich

Teste

- einzelne, positive Zahl
- einzelne, negative Zahl. Wird als 0-2 gewertet.

Output: `bool test_single_numbers()`

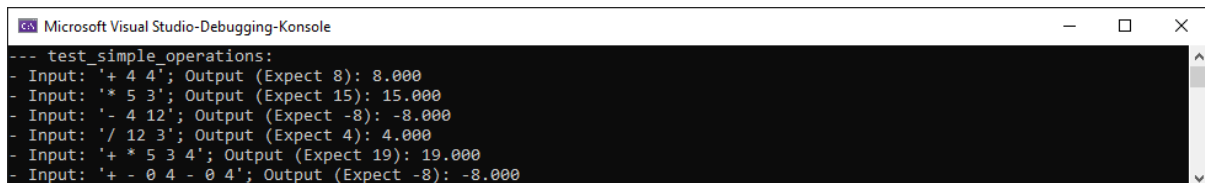


```
Microsoft Visual Studio-Debugging-Konsole
--- test_single_numbers:
- Input: '6'; Output (Expect 6): 6.000
- Input: '-2'; Output (Expect -2): Error: Error parsing 'Expression'.
```

Fall 4: Test erfolgreich

Teste diverse Rechenoperationen:

- Add
- Mult
- Sub
- Div
- Mehrere Operationen
- Addition zweier negativer Zahlen.


Output: `bool test_simple_operations()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_simple_operations:
- Input: '+ 4 4'; Output (Expect 8): 8.000
- Input: '* 5 3'; Output (Expect 15): 15.000
- Input: '- 4 12'; Output (Expect -8): -8.000
- Input: '/ 12 3'; Output (Expect 4): 4.000
- Input: '+ * 5 3 4'; Output (Expect 19): 19.000
- Input: '+ - 0 4 - 0 4'; Output (Expect -8): -8.000
```

Fall 5: Test erfolgreich

Teste auf

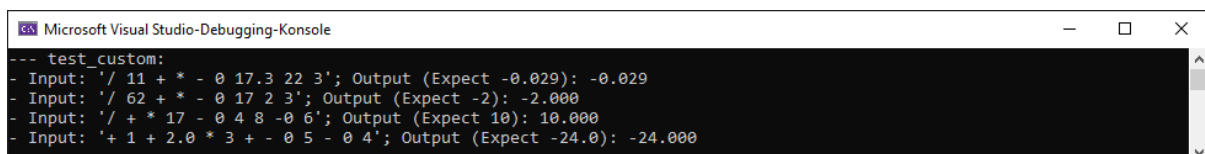
- Buchstaben (keine Variablen)
- Mehrere Operationen ohne Zusammenhang.

Output: `bool test_nonsense()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_nonsense:
- Input: '+ 3 c'; Output (Expect Caught Error): Error: Error parsing 'Expression'.
- Input: '* + * 3 3'; Output (Expect Caught Error): Error: Error parsing 'Expression'.
```

Fall 6: Test erfolgreich

Teste auf selbe Operationen wie in Fall 7 Beispiel 1. Hier ist anzumerken, dass es keine Klammern durch die Lesung der Präfix-Notation gibt. Die Tests werden dennoch mit aufgelistet als erweiterte Tests.

Output: `bool test_custom()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_custom:
- Input: '/ 11 + * - 0 17.3 22 3'; Output (Expect -0.029): -0.029
- Input: '/ 62 + * - 0 17 2 3'; Output (Expect -2): -2.000
- Input: '/ + * 17 - 0 4 8 -0 6'; Output (Expect 10): 10.000
- Input: '+ 1 + 2.0 * 3 + - 0 5 - 0 4'; Output (Expect -24.0): -24.000
```

Beispiel 3 Rechnen mit Variablen:

Lösungsidee

Der Code aus Beispiel 1 soll nun entsprechend erweitert werden, dass nicht nur Literale/Nummern, sondern auch vorher definierte Variablen verwendet werden können. Hierfür wurde die Grammatik entsprechend angepasst:

```
1  Expression = Term { AddOp Term }.
2  Term       = Factor { MultOp Factor }.
3  Factor      = [ AddOp ] ( Number | Variable | PExpression ).
4  PExpression = "(" Expression ")" .
5  AddOp       = "+" | "-" .
6  MultOp      = "*" | "/" .
7  Number      = Die Grammatik für Number ist im Scanner eingebaut (Integer | Real).
8  Variable    = Die Grammatik für Number ist im Scanner eingebaut (Identifizier).
```

Statt einer Nummer oder einer Expression in Klammern kann in einem Factor nun auch eine Variable vorkommen. Diese ist als Identifier durch den Scanner definiert und erkennbar.

Mithilfe einer öffentlichen Schnittstelle soll vorher durch eine Map ein oder mehrere Identifier mit je einer Nummer angegeben werden. Hierbei liegt es an dem Benutzer, der diese Map nun anlegt, ob Identifier doppelt zugewiesen werden können, dies soll nicht extra geprüft werden. Jedoch darf nur eine Nummer und kein Ausdruck zugewiesen werden. Diese Map soll dann genutzt werden, um die entsprechenden Variablen im Ausdruck umzuwandeln.

Es sollen sämtliche Tests wie in Beispiel 1 funktionieren, sowie sämtliche Ausnahmen ebenfalls berücksichtigt werden. Zudem soll es nicht möglich sein, Strings mit Leerzeichen als Variablen zu setzen, da diese nicht vom Scanner erkannt werden können. Es soll hier nicht geprüft werden, dass Variablen eine Zahl als erstes Zeichen haben können, obwohl dies ebenfalls vom Scanner nicht so erkannt wird, da dies spätestens beim parsen des Ausdrucks zu einem Fehler führen wird.

Testfälle

Fall 1: Test erfolgreich

Test auf Division durch Variable mit Wert 0.

Output: **bool** test_division_zero()

```
Microsoft Visual Studio-Debugging-Konsole
--- test_division_zero:
- Input: '((17 * -4) + 8) / x';
  Variables: { x:0.000 }
  Output (Expect Caught Error): Error: Error division by zero.
```

Fall 2: Test erfolgreich

Test auf einen Leeren Ausdruck mit gesetzter Variable.

Output: **bool** test_empty()

```
Microsoft Visual Studio-Debugging-Konsole
--- test_empty:
- Input: '';
  Variables: { x:1.000 }
  Output (Expect Caught Error): Error: Error parsing 'Expression'.
```

Fall 3: Test erfolgreich

Test auf

- nur eine Variable
- eine Variable mit einem minuszeichen davor. Es ginge ebenfalls eine negative Variable zu setzen s. Fall 7.

Output: **bool** test_single_numbers()

```
Microsoft Visual Studio-Debugging-Konsole
--- test_single_numbers:
- Input: 'x';
  Variables: { x:6.000 }
  Output (Expect 6): 6.000
- Input: '-x';
  Variables: { x:6.000 }
  Output (Expect -6): -6.000
```

Fall 4: Test erfolgreich

Test auf eine Variable, die im Ausdruck mehrfach verwendet wird.

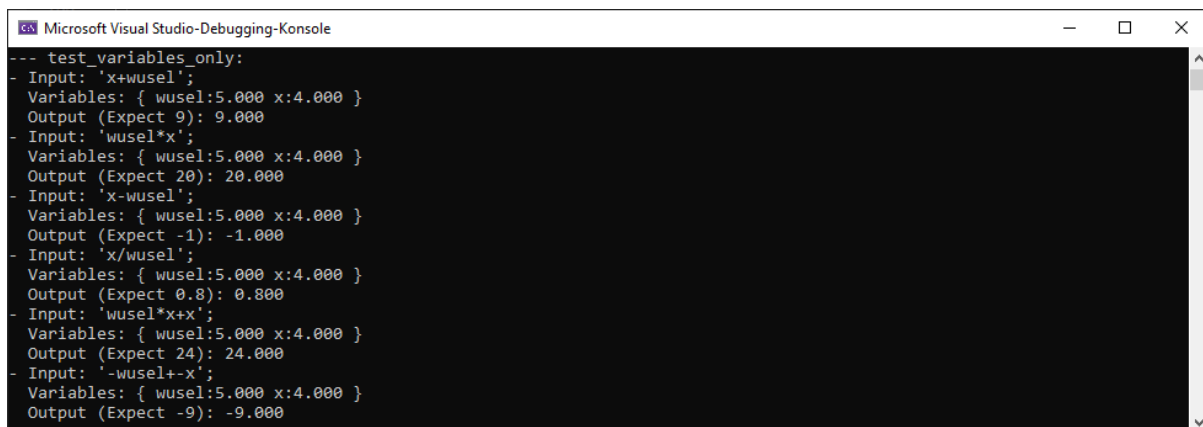
Output: **bool** test_multiple_usage()

```
Microsoft Visual Studio-Debugging-Konsole
--- test_multiple_usage:
- Input: 'x+x+x';
  Variables: { x:4.000 }
  Output (Expect 12): 12.000
```

Fall 5: Test erfolgreich

Test auf diverse Operationen nur mit Variablen

- Add
- Mult
- Sub
- Div
- Mehrere Operationen
- Addition von Variablen mit negativem Vorzeichen.

Output: `bool test_variables_only()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_variables_only:
- Input: 'x+wusel';
  Variables: { wusel:5.000 x:4.000 }
  Output (Expect 9): 9.000
- Input: 'wusel*x';
  Variables: { wusel:5.000 x:4.000 }
  Output (Expect 20): 20.000
- Input: 'x-wusel';
  Variables: { wusel:5.000 x:4.000 }
  Output (Expect -1): -1.000
- Input: 'x/wusel';
  Variables: { wusel:5.000 x:4.000 }
  Output (Expect 0.8): 0.800
- Input: 'wusel*x+x';
  Variables: { wusel:5.000 x:4.000 }
  Output (Expect 24): 24.000
- Input: '-wusel+-x';
  Variables: { wusel:5.000 x:4.000 }
  Output (Expect -9): -9.000
```

Fall 6: Test erfolgreich

Test auf Variablen innerhalb von Klammern.

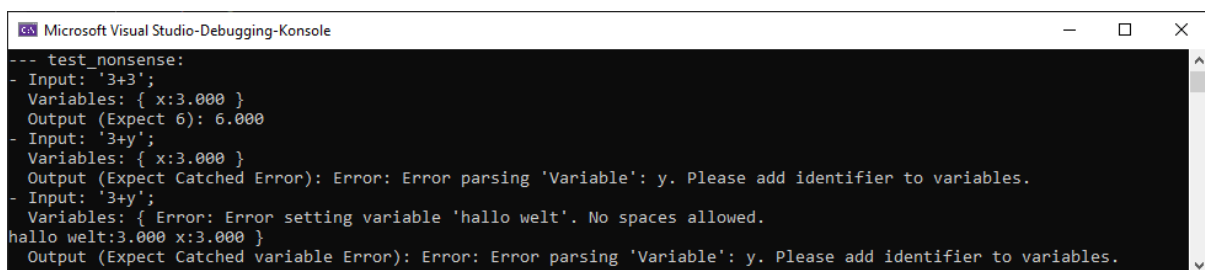
Output: `bool test_parenthesis()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_parenthesis:
- Input: '3*(x+5)';
  Variables: { x:4.000 }
  Output (Expect 27): 27.000
```

Fall 7: Test erfolgreich

Test auf

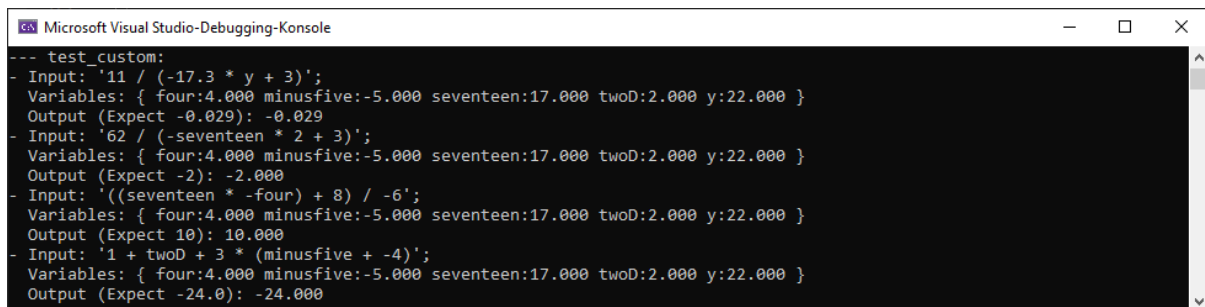
- einen Ausdruck weiterhin ohne Variablen
- einen Ausdruck mit unbekannter Variable
- einen String mit Leerzeichen als Variable. Hier werden zwei Error ausgegeben, einen bei der Zuweisung der Variable und einen beim Parsen des Ausdrucks, da „hallo“ und „welt“ nicht gefunden werden.

Output: `bool test_nonsense()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_nonsense:
- Input: '3+3';
  Variables: { x:3.000 }
  Output (Expect 6): 6.000
- Input: '3+y';
  Variables: { x:3.000 }
  Output (Expect Caught Error): Error: Error parsing 'Variable': y. Please add identifier to variables.
- Input: '3+y';
  Variables: { Error: Error setting variable 'hallo welt'. No spaces allowed.
hallo welt:3.000 x:3.000 }
  Output (Expect Caught variable Error): Error: Error parsing 'Variable': y. Please add identifier to variables.
```

Fall 8: Test erfolgreich

Testfälle von Fall 7 Beispiel 1 mit gemischten Tests und negativen Werten als Variable. Es werden dieselben Ergebnisse erwartet wie früher.

Output: `bool test_custom()`

```
Microsoft Visual Studio-Debugging-Konsole
--- test_custom:
- Input: '11 / (-17.3 * y + 3)';
  Variables: { four:4.000 minusfive:-5.000 seventeen:17.000 twoD:2.000 y:22.000 }
  Output (Expect -0.029): -0.029
- Input: '62 / (-seventeen * 2 + 3)';
  Variables: { four:4.000 minusfive:-5.000 seventeen:17.000 twoD:2.000 y:22.000 }
  Output (Expect -2): -2.000
- Input: '((seventeen * -four) + 8) / -6';
  Variables: { four:4.000 minusfive:-5.000 seventeen:17.000 twoD:2.000 y:22.000 }
  Output (Expect 10): 10.000
- Input: '1 + twoD + 3 * (minusfive + -4)';
  Variables: { four:4.000 minusfive:-5.000 seventeen:17.000 twoD:2.000 y:22.000 }
  Output (Expect -24.0): -24.000
```