

SWE-Übung 6

Zeitaufwand in h: 9

Inhaltsverzeichnis

Beispiel 1: Arithmetische Ausdrücke (infix).....	2
Lösungsidee	2
Code.....	2
Testfälle	3
Testfall 1: Test mit Files	3
Testfall 2: Einfache Normalfälle	3
Testfall 3: Punkt vor Strich.....	4
Testfall 4: Division durch 0	4
Testfall 5: Parse-Fehler mit eof	5
Testfall 6: Weitere Parse-Fehler	5
Beispiel 2: Arithmetische Ausdrücke (präfix)	6
Lösungsidee	6
Code.....	6
Testfälle	6
Testfall 1: Test mit Files	6
Testfall 2: Einfache Normalfälle	7
Testfall 3: Punkt vor Strich?.....	7
Testfall 4: Mehrere Zeichen hintereinander	8
Testfall 5: Division durch 0	8
Testfall 6: Sonderfälle und Probleme	9
Testfall 7: Ungültige Ausdrücke.....	9
Beispiel 3: Rechnen mit Variablen.....	10
Lösungsidee	10
Code.....	10
Testfälle	10
Testfall 1: Test mit Files	10
Testfall 2: Tests ohne Variablen	11
Testfall 3: Tests mit Variablen	11
Testfall 4: Normalfälle	12
Testfall 5: Division durch 0	12
Testfall 6: Ungültige Ausdrücke.....	13
Testfall 7: Keine Variablen bzw. Variable fehlt.....	13

Beispiel 1: Arithmetische Ausdrücke (infix)

Lösungsidee

Um die arithmetischen Ausdrücke parsen zu können, wird ein Scanner verwendet. Dieser wird immer zu Beginn des Parse-Vorgangs mit einem Eingabe-Stream initialisiert. Der Scanner erkennt einzelne (Sonder-) Zeichen, ganze (auch mehrstellige) Zahlen und Wörter, die aus beliebig vielen Buchstaben bestehen können.

Die Grammatik für den Parse-Vorgang war für dieses Beispiel bereits gegeben und sieht wie folgt aus. Das Bild wurde aus den Vorlesungs-Folien übernommen.

```
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“ .
```

Der Algorithmus beginnt somit immer mit einer Expression. Es folgt ein rekursiver Abstieg in Unterklassen, je nachdem welches Zeichen der Scanner liefert.

Der Parser prüft nicht nur die Korrektheit eines arithmetischen Ausdrucks, sondern liefert gleich dessen Ergebnis. Dabei muss auf einen Sonderfall geachtet werden: Bei einer Division durch Null soll eine Exception geworfen werden. Wenn sich ein Syntaxfehler im arithmetischen Ausdruck befindet, soll auch eine Exception geworfen werden mit der Information, bei welcher Klasse der Fehler aufgetreten ist.

Um etwas Code-Verdopplung bei den weiteren Beispielen zu vermeiden, wird der Scanner, die Definition der Exceptions und eine abstrakte Basis-Klasse des Parsers, der Grundfunktionen für alle Beispiele zur Verfügung stellt, außerhalb des Projekts abgelegt.

Die Basisklasse stellt unter anderem statische Methoden zur Ausgabe eines Files, das einen arithmetischen Ausdruck enthält, zur Verfügung.

Es muss beim Parsen einer Expression immer der komplette Stream abgearbeitet werden. Wenn aus irgendwelchen Gründen ein Ergebnis zurückgeliefert wurde, bevor alle Zeichen des Streams ausgelesen wurden, soll auch eine Parse-Exception geworfen werden. Diese Überprüfung soll auch in der Basisklasse erfolgen.

Code

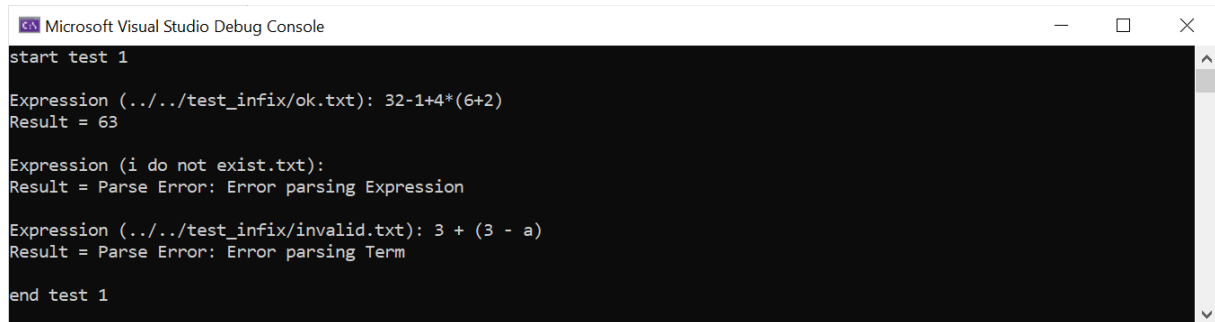
Der Code zu diesem Beispiel befindet sich in *Task1* in der Solution *SWE_Sandholzer_Ue06.sln*.

Testfälle

Testfall 1: Test mit Files

Bei diesem Test werden drei Files getestet.

Beim ersten File funktioniert alles wie erhofft, das zweite File existiert nicht und es wird ein Fehler ausgegeben, genauso wie beim dritten File, das eine Variable erhält.



```
Microsoft Visual Studio Debug Console

start test 1

Expression (../../test_infix/ok.txt): 32-1+4*(6+2)
Result = 63

Expression (i do not exist.txt):
Result = Parse Error: Error parsing Expression

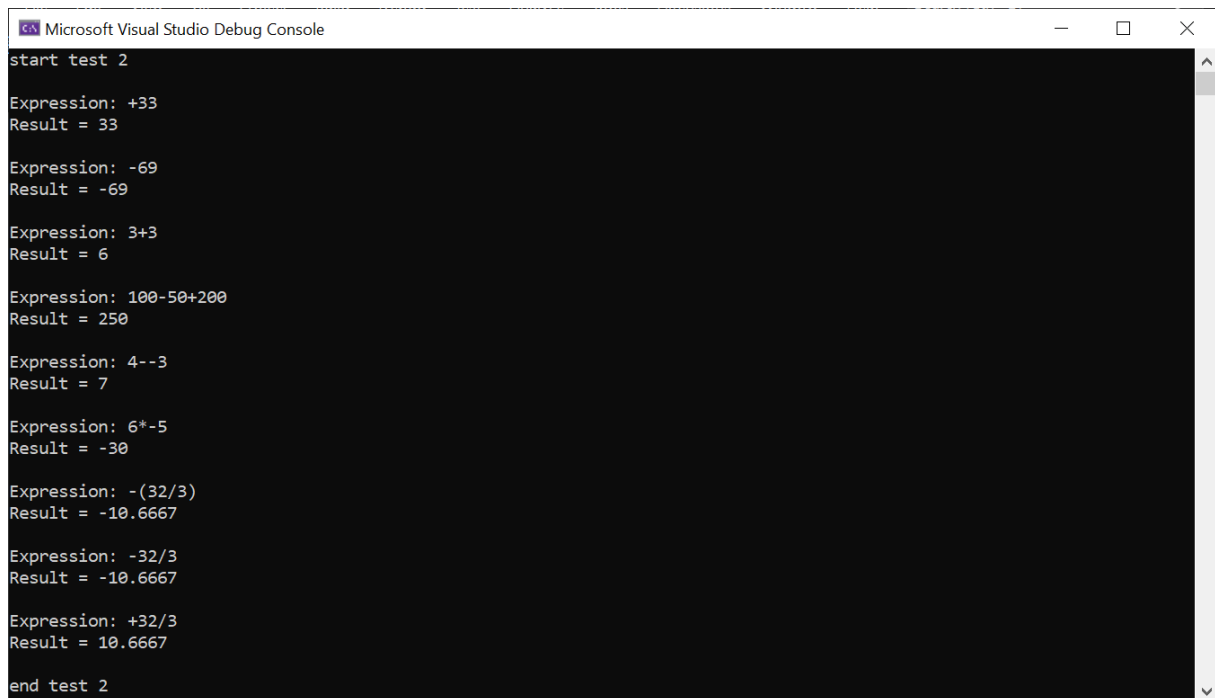
Expression (../../test_infix/invalid.txt): 3 + (3 - a)
Result = Parse Error: Error parsing Term

end test 1
```

Bei allen weiteren Testfällen werden statt Files String-Streams verwendet.

Testfall 2: Einfache Normalfälle

Besonders hervorzuheben ist hier der Umgang mit Vorzeichen. Vor jeder Zahl darf ein zusätzliches Plus oder Minus stehen. Dieses wird immer als Vorzeichen gewertet.



```
Microsoft Visual Studio Debug Console

start test 2

Expression: +33
Result = 33

Expression: -69
Result = -69

Expression: 3+3
Result = 6

Expression: 100-50+200
Result = 250

Expression: 4--3
Result = 7

Expression: 6*-5
Result = -30

Expression: -(32/3)
Result = -10.6667

Expression: -32/3
Result = -10.6667

Expression: +32/3
Result = 10.6667

end test 2
```

Testfall 3: Punkt vor Strich

Kla-Pu-Stri wird ordnungsgemäß befolgt!

```
Microsoft Visual Studio Debug Console

start test 3

Expression: 5+4*3
Result = 17

Expression: (5+4)*3
Result = 27

Expression: 5+(4*3)
Result = 17

Expression: (5+4*3)
Result = 17

Expression: (1-3/5)/8/10
Result = 0.005

Expression: (1-3)/5/8/10
Result = -0.005

Expression: 1-3/5/8/10
Result = 0.9925

end test 3
```

Testfall 4: Division durch 0

Division mit Null wird verhindert. Es ist auch interessant zu sehen, dass bei double-Werten -0 möglich ist. Semantisch ist der Wert trotzdem 0.

```
Microsoft Visual Studio Debug Console

start test 4

Expression: 1/0
Result = Parse Error: Divide by 0 Error!

Expression: 6+300000/(500*0)
Result = Parse Error: Divide by 0 Error!

Expression: 8/-0
Result = Parse Error: Divide by 0 Error!

Expression: -0/8
Result = -0

Expression: 0/69*-1
Result = -0

Expression: 0/69*-1*-1
Result = 0

Expression: 0*0*0/(6*4-24)
Result = Parse Error: Divide by 0 Error!

Expression: -0+3
Result = 3

Expression: 0+3
Result = 3

end test 4
```

Testfall 5: Parse-Fehler mit eof

Nachdem ein Ausdruck erfolgreich ausgewertet wurde, wird überprüft, ob der Stream bereits zur Gänze gelesen wurde. Ist dies nicht der Fall, werden Fehler ausgegeben (mit dem Ergebnis).

Bei anderen Fehlern, z.B. wenn der Stream endet, obwohl noch Zeichen erwartet werden, erscheint auch ein Fehler, je nachdem wo dieser aufgetreten ist.

```
Microsoft Visual Studio Debug Console

start test 5

Expression: 4,1 + 99
Result = Parse Error: Error finished parsing Expression (with result = 4.000000) before eof

Expression: 5 6
Result = Parse Error: Error finished parsing Expression (with result = 5.000000) before eof

Expression: 4(4+2)
Result = Parse Error: Error finished parsing Expression (with result = 4.000000) before eof

Expression: 5 *
Result = Parse Error: Error parsing Factor

Expression: (5*7
Result = Parse Error: Error parsing PExpression

Expression: +
Result = Parse Error: Error parsing Factor

Expression: 0-
Result = Parse Error: Error parsing Term

Expression: 78/3 3
Result = Parse Error: Error finished parsing Expression (with result = 26.000000) before eof

Expression: 1+1x
Result = Parse Error: Error finished parsing Expression (with result = 2.000000) before eof

end test 5
```

Testfall 6: Weitere Parse-Fehler

Weitere Spezialfälle. Ergebnisse wie erwartet. Erwähnenswert ist, dass auch Kommazahlen eingelesen werden können!

```
Microsoft Visual Studio Debug Console

start test 6

Expression:
Result = Parse Error: Error parsing Expression

Expression: 5-2)
Result = Parse Error: Error finished parsing Expression (with result = 3.000000) before eof

Expression: 7.1 - 3
Result = 4.1

Expression: 80/0.000001
Result = 8e+07

Expression: 4*x
Result = Parse Error: Error parsing Factor

Expression: +33
Result = 33

Expression: ++33
Result = Parse Error: Error parsing Factor

Expression: 3++3
Result = 6

Expression: 3+++3
Result = Parse Error: Error parsing Factor

end test 6
```

Beispiel 2: Arithmetische Ausdrücke (präfix)

Lösungsidee

Grundsätzlich läuft beim Präfix-Parser alles gleich ab wie schon beim Beispiel 1. Die Grammatik sieht nun aber anders aus:

AddOp und MultOp liefern wie üblich die Zeichen: + - und * /

Unsigned = Digit {Digit}

Dabei ist ein Digit eine Zahl zwischen 0 und 9, gleich wie beim Infix-Scanner.

Es werden zusätzlich nur noch folgende nicht-terminierende Klassen benötigt:

Expression = Unsigned | Sub-Expression

Sub-Expression = (AddOp | MultOp) (Unsigned | Sub-Expression) (Unsigned | Sub-Expression)

Jeder Ausdruck beginnt mit einem Operator (+, -, *, /), gefolgt zwei Operanden, diese können entweder aus einzelnen Zahlen (Unsigned) bestehen, oder aus einem weiteren Ausdruck, der wieder mit einem Operator beginnt. Eine Ausnahme bilden einzelne Zahlen, diese sollen auch als gültige Ausdrücke erkannt werden. Aus diesem Grund wird die Expression-Klasse so definiert, dass entweder ein arithmetischer Ausdruck (hier: Sub-Expression) oder eine einzelne Zahl erkannt werden kann.

Zahlen vom Stream sind IMMER positiv, also ein + oder ein – ist in jedem Fall ein Operator. Negative Zahlen müssen zum Beispiel über eine Subtraktion gebildet werden (-0 x). Es dürfen auch keine Klammern vorkommen. Die Reihenfolge der Operationen ist durch die Reihenfolge der Operatoren und Operanden eindeutig gegeben.

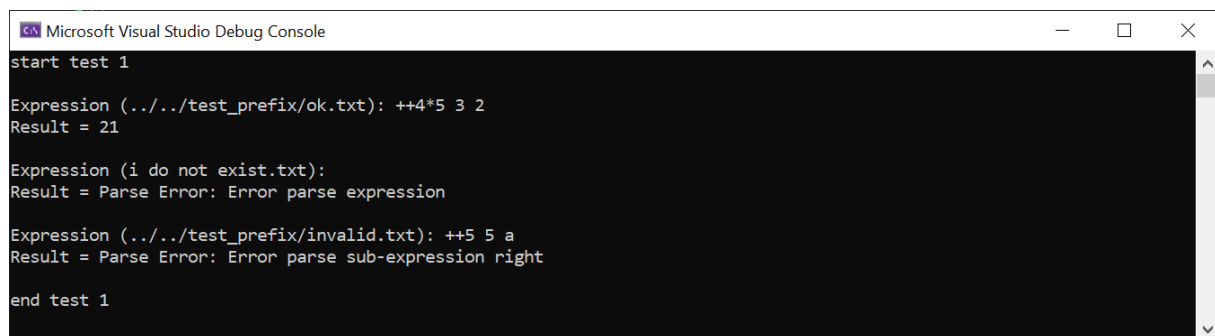
Code

Der Code zu diesem Beispiel befindet sich in *Task2* in der Solution *SWE_Sandholzer_Ue06*.

Testfälle

Testfall 1: Test mit Files

Einmal ein gültiges, nichtexistierendes und ein ungültiges File:



```
Microsoft Visual Studio Debug Console

start test 1

Expression (../../test_prefix/ok.txt): ++4*5 3 2
Result = 21

Expression (i do not exist.txt):
Result = Parse Error: Error parse expression

Expression (../../test_prefix/invalid.txt): ++5 5 a
Result = Parse Error: Error parse sub-expression right

end test 1
```

Ab sofort wird wieder mit String-Streams gearbeitet.

Testfall 2: Einfache Normalfälle

Hier werden die Tests vom zweiten Testfall des ersten Beispiels in Präfix-Notation getestet:

```
Microsoft Visual Studio Debug Console

start test 2

Expression: 33
Result = 33

Expression: -0 69
Result = -69

Expression: +3 3
Result = 6

Expression: +-100 50 200
Result = 250

Expression: -4-0 3
Result = 7

Expression: *6 -0 5
Result = -30

Expression: -0/32 3
Result = -10.6667

Expression: /32 3
Result = 10.6667

end test 2
```

Testfall 3: Punkt vor Strich?

Dritter Testfall von Beispiel 1 in Präfix-Notation:

```
Microsoft Visual Studio Debug Console

start test 3

Expression: +5 *4 3
Result = 17

Expression: *+5 4 3
Result = 27

Expression: *3+4 5
Result = 27

Expression: / / - 1 /3 5 8 10
Result = 0.005

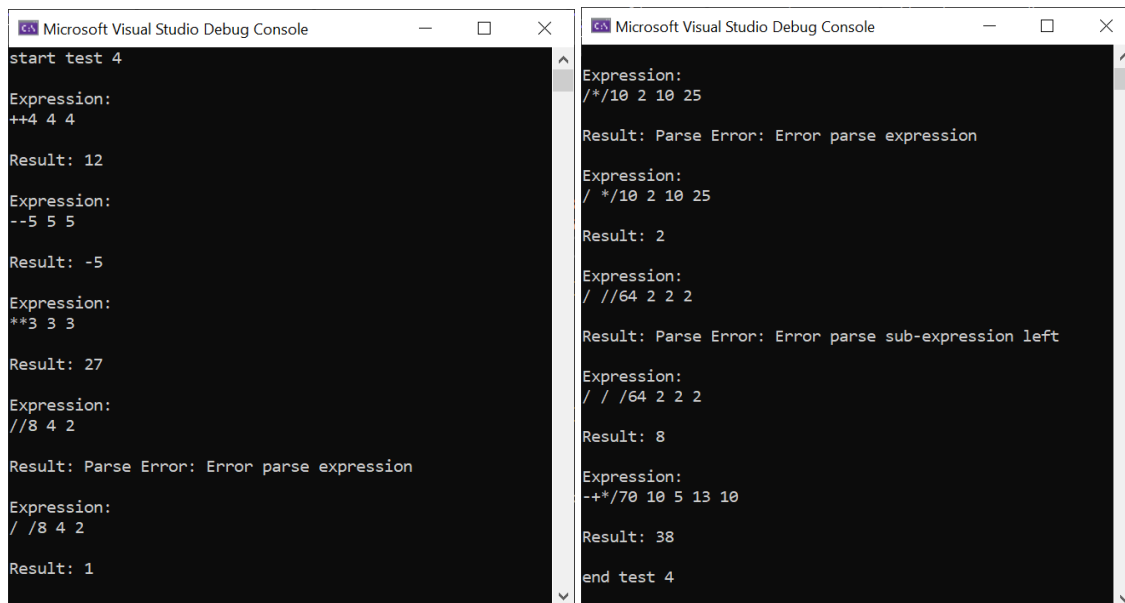
Expression: / / / - 1 3 5 8 10
Result = -0.005

Expression: - 1/ / / 3 5 8 10
Result = 0.9925

end test 3
```

Testfall 4: Mehrere Zeichen hintereinander

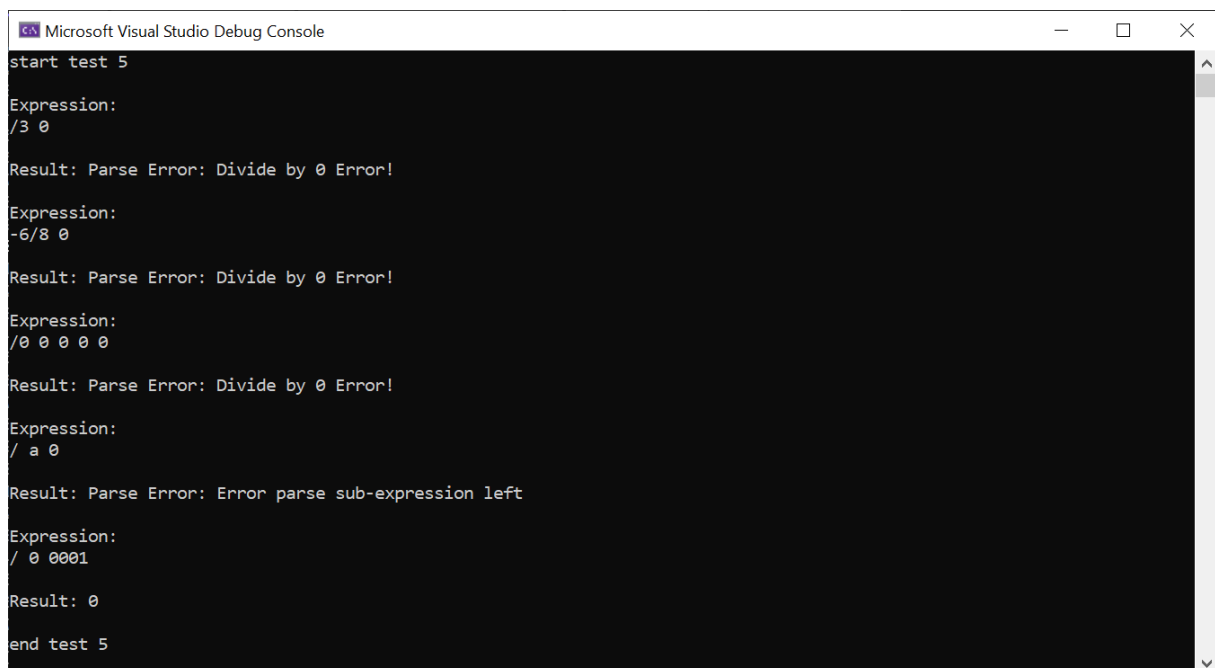
Beim Präfix-Scanner gibt es Probleme beim Dividiert-Zeichen! Danach darf kein weiterer Operator ohne Leerzeichen folgen, sonst wird es nicht als „MultOp“ erkannt.



```
Microsoft Visual Studio Debug Console
start test 4
Expression:
++4 4 4
Result: 12
Expression:
--5 5 5
Result: -5
Expression:
**3 3 3
Result: 27
Expression:
//8 4 2
Result: Parse Error: Error parse expression
Expression:
/ /8 4 2
Result: 1

Microsoft Visual Studio Debug Console
Expression:
/*10 2 10 25
Result: Parse Error: Error parse expression
Expression:
/ /*10 2 10 25
Result: 2
Expression:
/ //64 2 2 2
Result: Parse Error: Error parse sub-expression left
Expression:
/ / /64 2 2 2
Result: 8
Expression:
-+*/70 10 5 13 10
Result: 38
end test 4
```

Testfall 5: Divison durch 0



```
Microsoft Visual Studio Debug Console
start test 5
Expression:
/3 0
Result: Parse Error: Divide by 0 Error!
Expression:
-6/8 0
Result: Parse Error: Divide by 0 Error!
Expression:
/0 0 0 0 0
Result: Parse Error: Divide by 0 Error!
Expression:
/ a 0
Result: Parse Error: Error parse sub-expression left
Expression:
/ 0 0001
Result: 0
end test 5
```


Testfall 6: Sonderfälle und Probleme

Einzelne Zahlen werden richtig erkannt, allerdings nur ohne Zeichen. Sind zu viel oder zu wenig Operanden vorhanden, gibt's entsprechende Fehler.

```
Microsoft Visual Studio Debug Console

start test 6

Expression: 1000
Result: 1000

Expression: 0
Result: 0

Expression: -33
Result: Parse Error: Error parse sub-expression right

Expression: +34
Result: Parse Error: Error parse sub-expression right

Expression: +4 4
Result: 8

Expression: ++2 2 2
Result: 6

Expression: ++2 2 2 2
Result: Parse Error: Error finished parsing Expression (with result = 6.000000) before eof

Expression: +++2 2 2
Result: Parse Error: Error parse sub-expression right

end test 6
```

Testfall 7: Ungültige Ausdrücke

Wird immer richtig erkannt. Auch beim Präfix-Scanner sind Kommazahlen möglich.

```
Microsoft Visual Studio Debug Console

start test 7

Expression: (+1 1)
Result: Parse Error: Error parse expression

Expression: * 3 x
Result: Parse Error: Error parse sub-expression right

Expression: *
Result: Parse Error: Error parse sub-expression left

Expression:
Result: Parse Error: Error parse expression

Expression: hi
Result: Parse Error: Error parse expression

Expression: +5,5
Result: Parse Error: Error parse sub-expression right

Expression: +5.34 10
Result: 15.34

Expression: +-0 00 000
Result: 0

Expression: +3 3 nope
Result: Parse Error: Error finished parsing Expression (with result = 6.000000) before eof

end test 7
```

Beispiel 3: Rechnen mit Variablen

Lösungsidee

Bei diesem Beispiel sollen arithmetische Ausdrücke in Infix-Notation, die auch Variablen enthalten, ausgewertet werden. Dafür kann ein Großteil der Implementierung vom ersten Beispiel übernommen werden. Es ändert sich nur die Definition der Unsigned-Klasse:

Unsigned = (Digit {Digit} | Identifier {Identifier})

Dieser darf nun entweder aus einer Zahl oder einem Wort bestehen (min 1 Buchstabe).

Der Parser benötigt zum Rechnen mit Variablen eine Map aus Variablen mit den jeweiligen Werten. Diese muss dem Parser übergeben werden, bevor ein Ausdruck mit Variablen ausgewertet werden kann. Wenn bei der Auswertung unbekannte Variablen vorkommen, soll eine Exception geworfen werden.

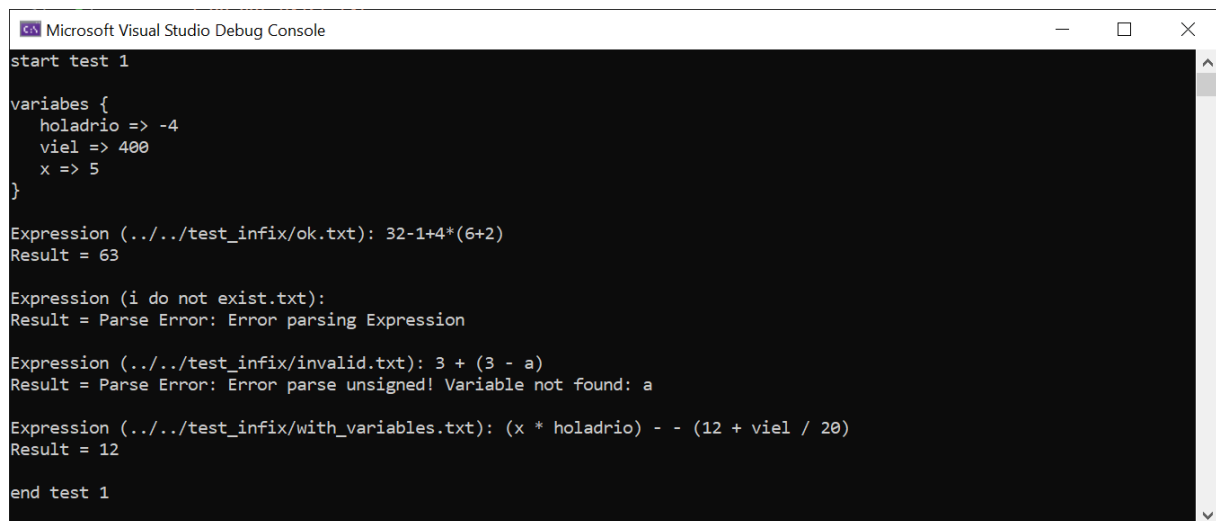
Code

Der Code zu diesem Beispiel befindet sich in *Task3* in der Solution *SWE_Sandholzer_Ue06.sln*.

Testfälle

Testfall 1: Test mit Files

Gleiche Files wie beim ersten Beispiel, plus eines mit Variablen die definiert wurden. Ergebnis wie erwartet. Die Variable ‚a‘ wurde nicht definiert, darum kommt bei diesem Ausdruck auch ein Fehler.



```
Microsoft Visual Studio Debug Console

start test 1

variables {
    holadrio => -4
    viel => 400
    x => 5
}

Expression (../../test_infix/ok.txt): 32-1+4*(6+2)
Result = 63

Expression (i do not exist.txt):
Result = Parse Error: Error parsing Expression

Expression (../../test_infix/invalid.txt): 3 + (3 - a)
Result = Parse Error: Error parse unsigned! Variable not found: a

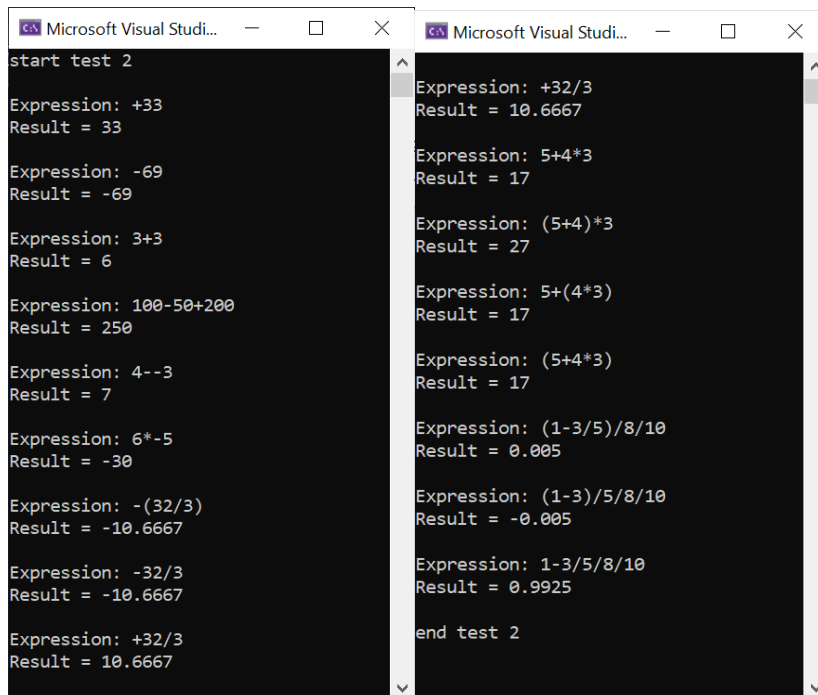
Expression (../../test_infix/with_variables.txt): (x * holadrio) - - (12 + viel / 20)
Result = 12

end test 1
```

Für die weiteren Testfälle werden wieder String-Streams verwendet.

Testfall 2: Tests ohne Variablen

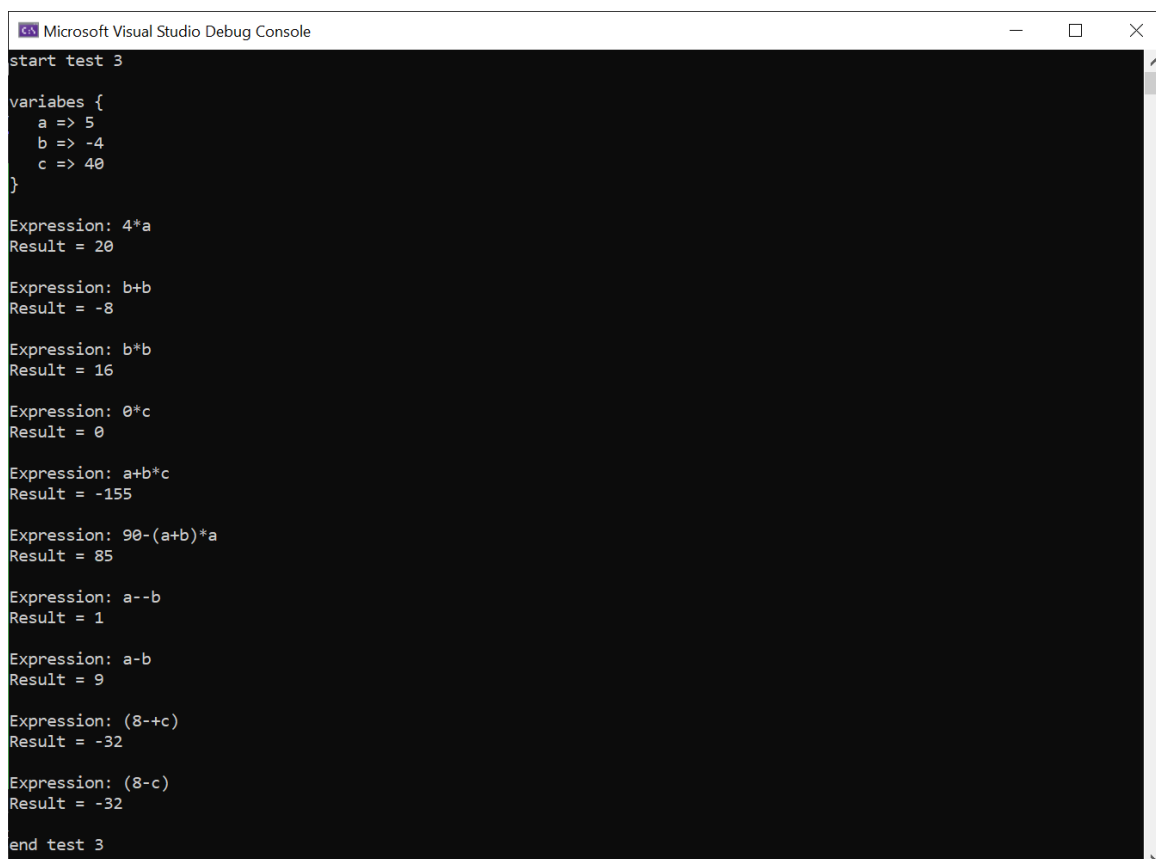
Zuerst sollen die Testfälle 2 und 3 von Beispiel 1 durchgeführt werden. So soll gezeigt werden, dass auch ohne Variablen alles wie erwartet funktioniert.



```
start test 2
Expression: +33
Result = 33
Expression: -69
Result = -69
Expression: 3+3
Result = 6
Expression: 100-50+200
Result = 250
Expression: 4--3
Result = 7
Expression: 6*-5
Result = -30
Expression: -(32/3)
Result = -10.6667
Expression: -32/3
Result = -10.6667
Expression: +32/3
Result = 10.6667
Expression: +32/3
Result = 10.6667
Expression: 5+4*3
Result = 17
Expression: (5+4)*3
Result = 27
Expression: 5+(4*3)
Result = 17
Expression: (5+4*3)
Result = 17
Expression: (1-3/5)/8/10
Result = 0.005
Expression: (1-3)/5/8/10
Result = -0.005
Expression: 1-3/5/8/10
Result = 0.9925
end test 2
```

Testfall 3: Tests mit Variablen

Da Variablen auch negative Werte annehmen können ist es möglich, dass z.B. $6 - - 3$ vorkommt, wie bei $a - -b$.



```
start test 3
variables {
  a => 5
  b => -4
  c => 40
}
Expression: 4*a
Result = 20
Expression: b+b
Result = -8
Expression: b*b
Result = 16
Expression: 0*c
Result = 0
Expression: a+b*c
Result = -155
Expression: 90-(a+b)*a
Result = 85
Expression: a--b
Result = 1
Expression: a-b
Result = 9
Expression: (8--c)
Result = -32
Expression: (8-c)
Result = -32
end test 3
```

Testfall 4: Normalfälle

Weitere Tests mit etwas komplizierteren Ausdrücken:

```
Microsoft Visual Studio Debug Console

start test 4

variables {
  a => 10
  b => -20
  c => 3
}

Expression: (a+b)*(a+c)
Result = -130

Expression: (a+b)*(a-b)
Result = -300

Expression: 100/a/c
Result = 3.33333

Expression: 100/a/c*c*c
Result = 30

Expression: (a+c)*((a-c)/(1-b))
Result = 4.33333

Expression: (((a+b)))
Result = -10

end test 4
```

Testfall 5: Division durch 0

Ein paar Tests mit 0, meist Divisionen durch 0, die verhindert werden.

```
Microsoft Visual Studio Debug Console

start test 5

variables {
  a => 10
  b => 0
}

Expression: 3/0
Result = Parse Error: Divide by 0 Error!

Expression: 5/(a-10)
Result = Parse Error: Divide by 0 Error!

Expression: 7-0/0
Result = Parse Error: Divide by 0 Error!

Expression: b
Result = 0

Expression: -b
Result = -0

Expression: 7/b
Result = Parse Error: Divide by 0 Error!

Expression: a*b+a/b
Result = Parse Error: Divide by 0 Error!

Expression: a*(b+a)/b
Result = Parse Error: Divide by 0 Error!

end test 5
```

Testfall 6: Ungültige Ausdrücke

Werden erkannt, sowohl wenn ein gültiges Ergebnis zurückgeliefert wird, aber der Stream noch nicht zu Ende ist, als auch, wenn die Syntax allgemein nicht passt.

```
Microsoft Visual Studio Debug Console

start test 6

variables {
  factor => 0
  term => 10
}

Expression: 5 6
Result = Parse Error: Error finished parsing Expression (with result = 5.000000) before eof

Expression: term(5+factor)
Result = Parse Error: Error finished parsing Expression (with result = 10.000000) before eof

Expression: 5 *
Result = Parse Error: Error parse factor

Expression: 1+1x
Result = Parse Error: Error finished parsing Expression (with result = 2.000000) before eof

Expression: term+4)
Result = Parse Error: Error finished parsing Expression (with result = 14.000000) before eof

Expression: 1++factor
Result = 1

Expression: 1+++factor
Result = Parse Error: Error parsing Factor

Expression: factor*
Result = Parse Error: Error parse factor

end test 6
```

Testfall 7: Keine Variablen bzw. Variable fehlt

Werden keine Variablen definiert, schlägt die Auswertung des Ausdrucks fehl. Je nachdem, ob und wo andere Fehler im Ausdruck vorkommen, kann auch eine andere Fehlermeldung ausgegeben werden. Beinhaltet ein Ausdruck keine Variablen, müssen auch keine definiert werden.

```
Microsoft Visual Studio Debug Console

start test 7

Expression: 3*x
Result = Parse Error: Error parse unsigned: variables expected!

Expression: (5*10)x
Result = Parse Error: Error finished parsing Expression (with result = 50.000000) before eof

Expression: var
Result = Parse Error: Error parse unsigned: variables expected!

Expression: 07*2+(5+8)*2
Result = 40

Expression: a*x*a*x
Result = Parse Error: Error parse unsigned: variables expected!

Expression:
Result = Parse Error: Error parsing Expression

Expression: 0
Result = 0

Expression: 7**4+value
Result = Parse Error: Error parse factor

end test 7
```