

Name: Nikolic Maja**Aufwand in h:** 12**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

Inhaltsverzeichnis	1
Beispiel 1 Infix.....	2
Lösungsidee	2
Quelltext: Infix.....	2
Tests	2
Beispiel 2 Präfix	3
Lösungsidee	3
Quelltext: Präfix.....	3
Tests	3
Beispiel 3 Rechnen mit Variablen.....	4
Lösungsidee	4
Quelltext: Rechnen mit Variablen	4
Tests	4

Beispiel 1 Infix

Lösungsidee

Für die Umsetzung dieses Beispiels wurde der Scanner benutzt, der uns inklusive Grammatik zur Verfügung gestellt wurde. Dieser ermöglicht es uns Symbole eines Streams einfach entgegenzunehmen und damit zu arbeiten, weil der Scanner schon Vorarbeit geleistet hat und unterschieden kann ob es sich bei einem Char um eine Zahl, Text bzw. Identifier oder Sonderzeichen handelt.

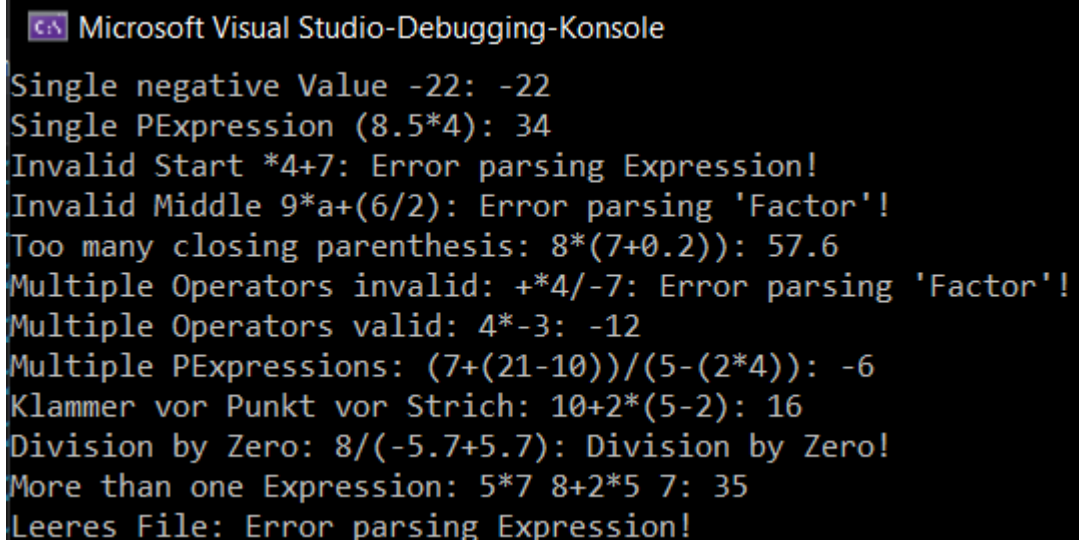
Auch die Grammatik für die Infix-Notation wurde uns zur Verfügung gestellt. Die Grammatik beschreibt den Aufbau eines Ausdrucks, den dieser erfüllen muss um erfolgreich ausgewertet zu werden.

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

Für jedes Non-Terminal-Symbol wird eine Methode angelegt, die überprüft, ob dieses tatsächlich entsprechend beginnt. Außerdem gibt es für jedes Non-Terminal-Symbol eine Methode welches das Symbol entsprechend der Grammatik parsed und den ermittelten Wert zurückgibt. Die Delegation, des Parsens auf andere Non-Terminal-Symbole wird rekursiver Abstieg genannt.

Quelltext: Infix

Tests



```
Microsoft Visual Studio-Debugging-Konsole
Single negative Value -22: -22
Single PExpression (8.5*4): 34
Invalid Start *4+7: Error parsing Expression!
Invalid Middle 9*a+(6/2): Error parsing 'Factor'!
Too many closing parenthesis: 8*(7+0.2)): 57.6
Multiple Operators invalid: +*4/-7: Error parsing 'Factor'!
Multiple Operators valid: 4*-3: -12
Multiple PExpressions: (7+(21-10))/(5-(2*4)): -6
Klammer vor Punkt vor Strich: 10+2*(5-2): 16
Division by Zero: 8/(-5.7+5.7): Division by Zero!
More than one Expression: 5*7 8+2*5 7: 35
Leeres File: Error parsing Expression!
```

Beispiel 2 Präfix

Lösungsidee

Auch für diese Beispiel wird der zur Verfügung gestellt Scanner verwendet. Allerdings ist die Grammatik für die Präfix Notation deutlich einfacher, da hier keine Klammern verwendet werden müssen, man nicht auf die Einhaltung von der Klammer-vor-Punkt-vor Strich-Regel achten muss und keine negativen Zahlen dargestellt werden können.

Die Grammatik, die ich für die Präfix-Notation erstellt habe, sieht folgendermaßen aus:

Expression = ((Operator Expression Expression) Number)
--

Operator = ,+' ,-' ,*' ,/'

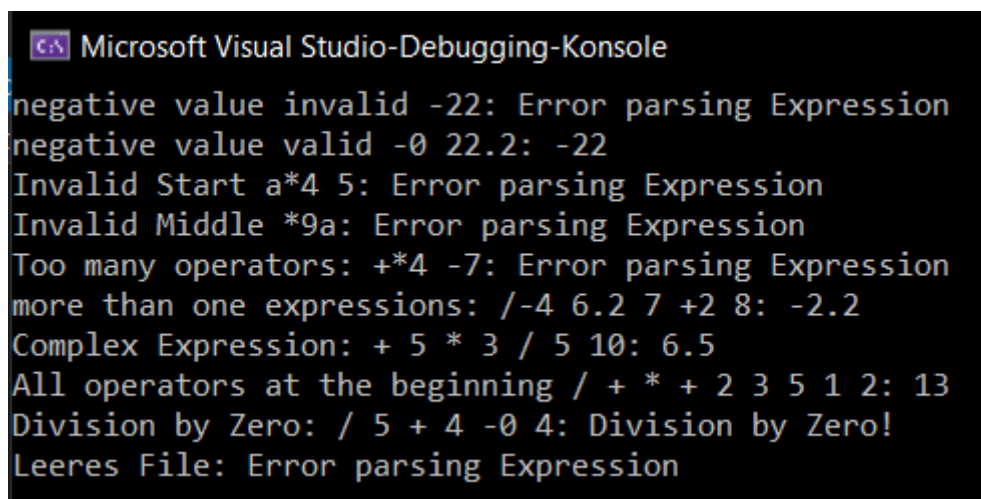
Eine Expression besteht demnach also immer entweder aus einem Operator, und zwei aufeinanderfolgen Expressions oder einer einfachen Zahl.

Number ist ein Non-Terminal-Symbol des Scanners und wird von ihm eigenständig erkannt.

Deshalb ist für das Parsen auch nur eine einzige Methode notwendig, da eine Expression selbst aus 2 Expressions besteht (oder einer einzelnen Zahl), welche rekursiv geparsed werden. Es ist lediglich eine Unterscheidung zwischen den Operatoren nötig, welche die Expressions entsprechend verknüpft.

Quelltext: Präfix

Tests



```
Microsoft Visual Studio-Debugging-Konsole
negative value invalid -22: Error parsing Expression
negative value valid -0 22.2: -22
Invalid Start a*4 5: Error parsing Expression
Invalid Middle *9a: Error parsing Expression
Too many operators: +*4 -7: Error parsing Expression
more than one expressions: /-4 6.2 7 +2 8: -2.2
Complex Expression: + 5 * 3 / 5 10: 6.5
All operators at the beginning / + * + 2 3 5 1 2: 13
Division by Zero: / 5 + 4 -0 4: Division by Zero!
Leeres File: Error parsing Expression
```

Beispiel 3 Rechnen mit Variablen

Lösungsidee

Um die Verwendung von Variablen zu ermöglichen wird die Parser-Klasse aus dem 1. Beispiel um eine weitere Datenkomponente, eine Map, ergänzt. Diese Map enthält die Variablennamen als Key und den jeweiligen Wert als Value und kann beim Konstruktor-Aufruf übergeben werden.

Der Scanner hat auch eine eigene Grammatik implementiert, wenn es sich um einen solchen Variablennamen (Identifizier) handelt:

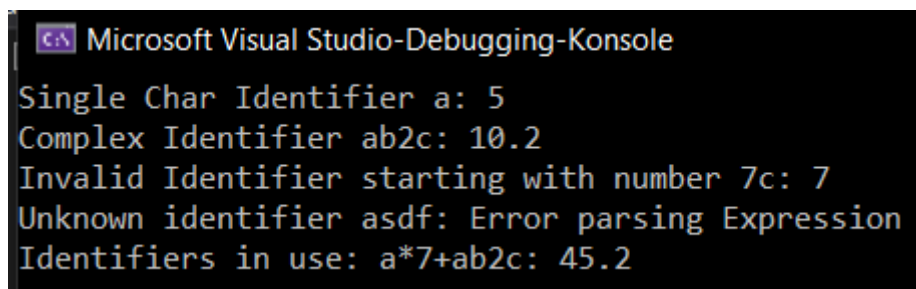
```
Identifizier = Alpha { Alpha | Digit } .
```

Die Grammatik für die Infix-Notation wird nun so abgeändert, dass ein Faktor aus einem optionalen AddOp und entweder einem Unsigned, einer PExpression oder einem solchen Identifizier besteht.

Wird nun ein Identifizier vom Parser erkannt, so muss zuerst überprüft werden ob sich der Variablenname überhaupt in der Map befindet, falls dies der Fall ist so wird der entsprechende Value aus der Map zurückgegeben, ansonsten wird ein Fehler geworfen.

Quelltext: Rechnen mit Variablen

Tests



```
Microsoft Visual Studio-Debugging-Konsole
Single Char Identifizier a: 5
Complex Identifizier ab2c: 10.2
Invalid Identifizier starting with number 7c: 7
Unknown identifizier asdf: Error parsing Expression
Identifiziers in use: a*7+ab2c: 45.2
```