SWE3 Übung 06

1. Arithmetische Ausdrücke in Infix-Notation

1.1 Lösungsidee

Die Grammatik sowie die Implementierung für arithmetische Ausdrücke wurde bereits im Rahmen der Übung großteils umgesetzt.

Bei der Erstellung eines Parsers kommt es im Wesentlichen auf die korrekte Interpretation der vorliegenden Grammatik an. Diese wird üblicherweise in EBNF-Schreibweise notiert und besteht aus kaskadierenden Elementen, die auch als Non-Terminale Symbole bezeichnet werden. Diese können, wie die Bezeichnung bereits nahelegt, nicht weiter zerlegt werden. Beispiele dafür sind Digits, also Zahlen, oder binäre Operationen wie Addition, Subtraktion, etc. Die Non-Terminalen Symbole werden in Gruppen zu den Terminal-Symbolen zusammengefasst und bilden dadurch vollständige Sätze einer Grammatik.

Die Grammatik der arithmetischen Ausdrücke gestaltet sich wie folgt:

```
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“ .
```

Für Non-Terminale Symbole werden bei der Implementierung immer zwei Methoden benötigt: einerseits die Prüfung, ob die entsprechende Gruppe vorliegt, und das Parsen selbst. Zur Gliederung der Grammatik kommen insgesamt acht verschiedene Metazeichen zum Einsatz:

- | … Alternative, „oder“
- . … Abschluss einer Regel / eines Satzes
- { } … Multiplizität
- [ ] … Optionalität
- ( ) … Gruppierung

1.2 Quelltext

Ein Parser wird nach dem sog. Rekursivem Abstieg programmiert: Jeder Methodenaufruf eines Non-Terminalen Symbols delegiert weiter zur darunterliegenden Klasse, bis schließlich das terminale Symbol geprüft wird. So wird beispielsweise bei einer Expression geprüft, ob es sich um einen Term handelt, da eine Expression laut Grammatik mit einem Term beginnen muss. Bei der Prüfung auf einen Term wird geprüft, ob es sich um einen Faktor handelt, da ein Term mit einem Faktor initialisiert wird. Ein Faktor wiederum kann entweder aus einer optionalen AddOp-Klasse bestehen oder aus einem Unsigned oder einem Klammerausdruck. Da hier alle drei Ausdrücke möglich sind, werden diese mit dem logischen Oder verknüpft. Dieses Vorgehen wird als rekursiver Abstieg bezeichnet.

Beim Parsen werden die Regeln der Grammatik umgesetzt. Zu Beginn jeder Parse-Methode wird geprüft, ob es sich beim anliegenden Zeichen im Scanner auch tatsächlich um die richtige Klasse handelt und ggf. eine Exception ausgelöst.

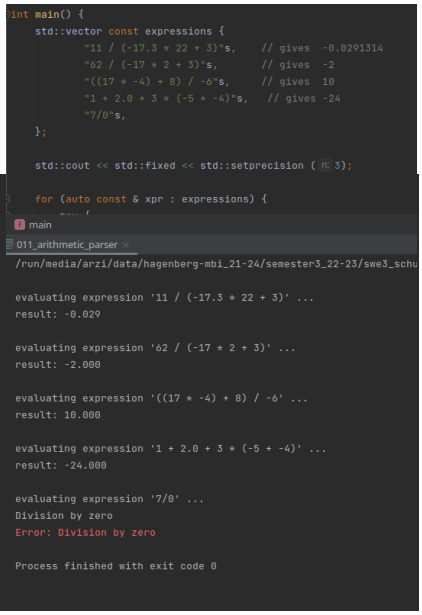
Je nach Metazeichen sind verschiedene Programmstrukturen notwendig – eine Optionalität kann etwa durch eine if-Abfrage geprüft werden, eine Multiplizität durch eine Schleife und eine Oder-Verknüpfung durch eine gestaffelte if-else-Abfrage.

Werte werden niemals direkt vom Scanner abgefragt, sondern über die entsprechenden Parse-Methoden retourniert.

Ein semantisches Problem beim Scanner entsteht, wenn ein korrekter Ausdruck gelesen wird, und vor Ende des Ausdrucks ein ungültiger Wert auftritt wie beispielsweise eine zusätzliche geschlossene Klammer. Hier bricht der Scanner ab und gibt das bis dahin berechnete Ergebnis zurück, obwohl nicht der gesamte Input gelesen wurde. Dieses Problem wurde in der Vorlesung mit Hr. Prof. Kulcycki diskutiert.

1.3 Testfälle

Ausgabe der vorgefertigten Testfälle von Hr. Prof. Kulcycki und Ergänzung um die Division durch Null:



Weitere Testfälle:

```
"7 / 0"s,                // division by zero
"- 4 + 4"s,              // start expression with negative-sign
"( ( 5 / 2 )"s,          // invalid expression (missing closing parenthesis
"( 1 + 2 + 3 ) * 2 ) - 8"s,  // -8 is cut off due to closing p. without opening p.
"( ( (4 + 3 ) ) )"s,     // lots of useless parenthesis
"3 * - 2"s,              // leading minus as negative number
"4 + +"s,                // "increment" not valid by grammar rules
"1 4"s,                  // missing factor leads to cutting the trailing end
"1 + 2 * 3 - 2"s,        // gives 5, priority of factors set by grammar
"10 + 6 - ( )"s,         // invalid due to empty parenthesis
"75 a - 13"s,            // illegal character is ignored!
"45 * a +2"s,            // illegal use of variable
```

```
evaluating expression '7 / 0' ...
Division by zero

evaluating expression '- 4 + 4' ...
result: 0.000

evaluating expression '( ( 5 / 2 )' ...

evaluating expression '( 1 + 2 + 3 ) * 2 ) - 8' ...
result: 12.000

evaluating expression '( ( (4 + 3 ) ) )' ...
result: 7.000
```

```
evaluating expression '3 * - 2' ...
result: -6.000

evaluating expression '4 + +' ...

evaluating expression '1 4' ...
result: 1.000

evaluating expression '1 + 2 * 3 - 2' ...
result: 5.000

evaluating expression '10 + 6 - ( )' ...

evaluating expression '75 a - 13' ...
result: 75.000

evaluating expression '45 * a +2' ...
Error: Division by zero
Error: Expected 'right parenthesis' but have {{end of file,ts,7}}.
Error: Error parsing 'Factor'.
Error: Error parsing 'expression'.
Error: Error parsing 'Factor'.
```

Die einzelnen Testfälle sind als Kommentar kurz erklärt; jeder Testfall gibt das erwartete Ergebnis zurück. Vor allem anzumerken ist die korrekte Reihenfolge bei der Berechnung mit Strich- und Punktoperationen ohne explizite Klammerung. Hier wird die richtige Reihenfolge durch die Grammatik vorgegeben.

Aufwand in Stunden: 5

2. Arithmetische Ausdrücke in Präfix-Notation

2.1 Lösungsidee

Ähnlich wie zu Beispiel 1 soll diesmal eine Grammatik samt Parser für arithmetische Ausdrücke in Präfix-Notation entwickelt werden. Im Gegensatz zur Infixnotation (Beispiel 1) stehen die Operatoren nicht zwischen den Operanden, sondern davor. So ergibt der Ausdruck „A + B“ in Präfix-Notation „+ A B“, aus „A * B + C * D“ wird „+ * A B * C D“ etc.

Ein Ausdruck besteht also mindestens aus einem Operanden oder einem Operator gefolgt von zwei Operanden. Ein Operand ist dabei entweder eine Ziffer oder aber ein vollständiger Ausdruck. Ein Operator ist ein herkömmliches terminales mathematisches Symbol wie Addition oder Subtraktion.

Die entsprechende Grammatik lautet also:

```
Expression = Operator Operand Operand | Operand .
Operator = "+" | "-" | "*" | "/" .
Operand = Unsigned | Expression .
Unsigned = Digit { Digit } .
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

Bevor die Grammatik als Parser implementiert wird, sollten die folgenden exemplarischen Ausdrücke in Präfix-Notation geprüft werden, ob sie der Grammatik entsprechen (mit dem Wissen, dass diese Ausdrücke korrekt sind):

+ A B	Operator Operand Operand → korrekt
+ A * B C	Operator Operand *Expression* Operand Operand → korrekt
* + A B C	Operator *Expression* Operand Operand Operand → korrekt
+ + A * B C D	Operator *Expression* Operand Expression Operand Operand Operand → korrekt
* + A B + C D	Operator *Expression* Operand Operand Operator Operand Operand → korrekt
+ * A B * C D	Operator *Expression* Operand Operand Operator Operand Operand → korrekt
+ + + A B C D	Operator *Expression* Operator Operand Operand Operand Operand → korrekt

(Wurde ein Operand als Expression erkannt, wird kursiv-Druck verwendet.)

2.2 Quelltext

Die Implementierung erfolgt auch hier wieder unter Verwendung des rekursiven Abstiegs: Jedes Non-Terminale Symbol wird geprüft und geparst. Zusätzlich gibt es zur Berechnung der Zwischenergebnisse eine Hilfsmethode *calc()*, die drei Parameter übernimmt: die beiden Operanden sowie den Operator als String. Dies dient der besseren Lesbarkeit des Codes, die Rechenoperation könnte auch direkt inline durchgeführt werden.

Liegt am Scanner ein Operator an, so werden die nächsten beiden Ausdrücke mit der *parseOperand()*-Methode eingelesen und der *calc*-Methode übergeben. Wird der *parseOperand()*-Methode jedoch eine Expression übergeben anstatt eines Operanden, so wird erneut die *parseExpression()* aufgerufen, ähnlich einem „rekursiven“ Aufruf. Bei korrekter Syntax „wickelt“ sich die *parseExpression()* vollständig aus und übernimmt schlussendlich die ursprünglichen Werte für die Berechnung des ersten Aufrufs.

Da bei der *calc()*-Methode der Aufruf auf *parseOperator()* einen String zurückliefert, wird hier mit dem bereits bekannten if-else-Verzweigung die korrekte Rechenoperation ausgewählt. Alternativ wäre hier auch ein Switch-Case möglich, wenn der Operator als Char übergeben wird.

2.3 Testfälle

Die folgenden Testfälle wurden ausgeführt; diese decken auch die in der Lösungsidee beschriebenen Ausdrücke ab:

```
"+ 1 2"s,              // gives 3, just basic test
"+ + + 1 2 3 4"s,      // gives 10; 1 + 2 + 3 + 4
"+ * 2 3 * 4 5"s,      // gives 26; 2 * 3 + 4 * 5
"* + 2 3 + 4 5"s,      // gives 45; ( 2 + 3 ) * ( 4 + 5 )
"/ 1 3"s,              // gives a third; 1 / 3
"/ 5 0"s,              // division by zero
"+ 2 * 3 4"s,          // gives 14; 2 + 3 * 4
"- 5 6"s,              // gives -1; 5 - 6
```

```
evaluating expression '+ 1 2' ...
result: 3

evaluating expression '+ + + 1 2 3 4' ...
result: 10

evaluating expression '+ * 2 3 * 4 5' ...
result: 26

evaluating expression '* + 2 3 + 4 5' ...
result: 45

evaluating expression '/ 1 3' ...
result: 0.333333

evaluating expression '/ 5 0' ...

evaluating expression '+ 2 * 3 4' ...
result: 14

evaluating expression '- 5 6' ...
result: -1
Error: Division by zero not allowed.

Process finished with exit code 0
```

Im Gegensatz zur Infixnotation können hier keine negativen Zahlen übergeben werden, da das Vorzeichen-Minus als Operator interpretiert werden würde. Laut Online-Recherche wird die Berechnung von negativen Zahlen durch die Ergänzung um „0 – *number*“ realisiert.

Hier führen doppelte Operatoren (die als Vorzeichen gedacht wären) zu einer SIGSEGV. Eine Implementierung ist aus zeitlichen Gründen (Klausurphase) leider nicht gelungen.

Aufwand in Stunden: 12

3. Arithmetische Ausdrücke in Infix-Notation mit Variablen

3.1 Lösungsidee

Dieses Beispiel stellt eine Erweiterung aus Aufgabe 1 dar, da die bestehende Infixnotation nun um Variablen erweitert werden soll. Die Variablen müssen vor Berechnung bekannt sein und dürfen nur terminale Ausdrücke enthalten, also beispielsweise *x = 42*, aber nicht *x = y + 7*.

Die Grammatik muss entsprechend erweitert werden, da anstatt eines Faktors nun auch eine Variable auftreten kann:

```
Expression = Term { AddOp Term } .
Term = Factor { MultOp Factor } .
Factor = [ AddOp ] ( Unsigned | PExpression | Variable) .
Unsigned = Digit { Digit } .
PExpression = ”(“ Expression ”)“ .
Variable = ” variableMap ” .  // variableMap as extern map-storage
AddOp = ”+“ | ”−“ .
MultOp = ”·“ | ”/“ .
Digit = ”0“ | ”1“ | ”2“ | ”3“ | ”4“ | ”5“ | ”6“ | ”7“ | ”8“ | ”9“ .
```

Eine Variable ist in einer externen Speicherstruktur vorhanden; der Ausdruck variableMap stellt somit ein terminales Symbol dar und ist daher unter Anführungszeichen gestellt.

3.2 Quelltext

Da Variablen im Wesentlichen die Definition eines *Key-Value-Pairs* darstellen, bietet sich die *std::map* ausgesprochen gut an. So werden in einer Map als *Key* der Variablenname und als *Value* der Wert der Variable gespeichert. Die Grammatik wird um den Begriff der Variable erweitert; beim Parsen greift der Algorithmus auf die Map zurück und liefert den entsprechen *Value*. Ähnlich bei der Prüfung, ob es eine Variable gibt, also ob der Ausdruck gültig ist: Die *Map* wird mithilfe von Iteratoren durchlaufen, und wenn die gesuchte Variable enthalten ist, wird *true* zurückgegeben.

Da die Variablen vor Durchführung der Berechnungen angelegt werden müssen, wird eine Hilfsmethode aufgerufen, mit der Variablen angelegt und gespeichert werden können:

```
 ArithmeticVariableParser().setVariable("a", -1);
ArithmeticVariableParser().setVariable("b", 3);
ArithmeticVariableParser().setVariable("hugeValue", 100);
ArithmeticVariableParser().setVariable("x", 42);
ArithmeticVariableParser().setVariable("y", 5);
ArithmeticVariableParser().setVariable("ZERO", 0);
ArithmeticVariableParser().setVariable("ONE", 1);
```

3.3 Testfälle

Tests ohne Variablen mit gültigen Eingaben:

```
evaluating expression '- 4 + 4' ...
result: 0.000

evaluating expression '( 1 + 2 + 3 ) * 2 ) - 8' ...
result: 12.000

evaluating expression '( ( (4 + 3 ) ) )' ...
result: 7.000

evaluating expression '3 * - 2' ...
result: -6.000

evaluating expression '1 4' ...
result: 1.000

evaluating expression '1 + 2 * 3 - 2' ...
result: 5.000
```

Tests unter Verwendung der oben gespeicherten Variablen:

```
"a"s,                    // gives the value of a which is -1
"hugeValue + 31 / 2"s,       // gives 115.5: 100 + 31 / 2
"2 * a - 1"s,              // gives -3: 2 * -1 - 1
"3 + -b + 2"s,             // gives 2: 3 + (-3) + 2
"5 * z"s,                  // error as z does not exist
"10  / ZERO"s,             // division by zero - invalid
"a * b + hugeValue - ONE"s,    // gives 96: -1 * 3 + 100 - 1
```

```
evaluating expression 'a' ...
result: -1.000

evaluating expression 'hugeValue + 31 / 2' ...
result: 115.500

evaluating expression '2 * a - 1' ...
result: -3.000

evaluating expression '3 + -b + 2' ...
result: 2.000

evaluating expression '5 * z' ...

evaluating expression '10  / ZERO' ...
Division by zero

evaluating expression 'a * b + hugeValue - ONE' ...
result: 96.000
Error: Variable 'z' is not defined.
Error: Division by zero

Process finished with exit code 0
```

Wie gezeigt, sind Rechenoperationen mit zuvor definierten Variablen möglich, auch wenn die Rechnung ausschließlich aus Variablen besteht.

Auf Fehler wie Division durch Null oder ungültige Variable wird entsprechend reagiert.

Aufwand in Stunden: 4