

05.02.2023

SWE Übung 06


s2010458016

MARCO SARKADY

WS 22/23

GESAMTAUFWAND:10 STUNDEN

Medizin – und Bioinformatik



Beispiel 1:

Lösungsidee:

In diesem Beispiel sollten wir einen Algorithmus erstellen, welcher eine Rechnung in der uns bereits gängigen Infix-Notation auszurechnen. Dafür haben wir eine Grammatik zur Verfügung gestellt bekommen, welche wir dann mit mehreren Funktionen und Methoden implementieren können.

Mit dieser Grammatik und der Hilfe der pfc-scanner-Implementierung kann man nun Funktionen zur Bestimmung der Art einer Rechnung erstellen. Die Implementierung geht dafür vom Äußersten zum Innersten.

```
Expression = Term { AddOp Term } .  
Term       = Factor { MultOp Factor } .  
Factor     = [ AddOp ] ( Unsigned | PExpression ) .  
PExpression = „ ( “ Expression „ ) “ .  
AddOp      = „+“ | „-“ .  
MultOp     = „*“ | „/“ .
```

Zuerst wird überprüft, ob die Rechnung eine Expression ist. Dadurch wird dann die Funktion aufgerufen, welche überprüft, ob es sich um einen Term handelt, dann um einen Faktor und so weiter.

Danach brauchen wir Funktionen, welche durch diese Teile Parsen, und den Mathematischen Ausdruck richtig Teilen und ausrechnen. Man geht hier also iterativ immer weiter die Begriffs-Hierarchie hinunter, bis man an den normalen Rechnungsteilen AddOp und MultOp angelangt ist.

Testfälle:

```
Test 1: Test mit mehreren Rechnungen:  
(17 * 22 + 3) / 11 = 34.273  
(-17 * 2 + 3) / 62 = -0.500  
(17 * 4) + 8 / 6 = 69.333  
1 + 2.0 + 3 * (-5 + -4) = -24.000
```

```
Test 2: Test mit negativen Zahlen:  
-5 * -3.5 - (-5 + -2.33) = 24.830  
(8 * -13) / 22 = -4.727
```

```
Test 3: Test mit falscher Grammatik:  
Error: Error parsing `Expression`.
```

```
Test 4: Test mit Variablen:  
Error: Error parsing `Term`.  
Error: Error parsing `Expression`.  
Error: Error parsing `Term`.  
Error: Error parsing `Term`.
```

```
Test 5: Test mit leeren strings:  
Error: Error parsing `Expression`.  
Error: Error parsing `Expression`.  
Error: Error parsing `Expression`.  
Error: Error parsing `Expression`.
```

```
Test 6: Test mit Division durch 0:  
Error: Error trying to divide by zero!  
Error: Error trying to divide by zero!
```

```
Test 7: Test with wrong paranthesis:  
Error: Expected 'right parenthesis' but have {{end of file,ts,7}}.
```

Beispiel 2:

Lösungsidee:

Dieses Beispiel ist ähnlich wie das erste, nur dass wir uns hier mit einer sogenannten PräfixNotation beschäftigen. Dies bedeutet, dass sich nun die Rechenzeichen am Beginn des

Ausdrucks befinden, und die Zahlen je nach ihrer Originalen Position in der Infix-Notation dahinter gereiht sind. Dazu kann man im Grunde sagen, dass nach einem Rechenzeichen immer zwei Zahlen stehen müssen, um sie auszurechnen.

Wie zum Beispiel:

$2 + 3 = 5$ wird zu: $+ 2 3 = 5$

$6 / (4 * 3) = 2$ wird zu: $/ 6 * 4 3 = 2$

Expression =	Unsigned (BinaryOp Term Term)
BinaryOp =	AddOp MultOp
Term =	Expression
MultOp =	* /
AddOp =	+ -
Unsigned =	Digit {Digit} -> 1 oder 10
Digit =	0 1 2 3 4 5 6 7 8 9

Was bisher noch nicht so gut funktioniert, ist Test mit falscher Grammatik. Da bekomme ich Ergebnisse, obwohl man ein parsing Error erwartet.

Testfälle:

```
Test 1: Test mit normalen Rechnungen:  
+ 2 3 = 5.000  
/ 6 3 = 2.000  
* 5 2 = 10.000  
- 2 7 = -5.000  
* 7 + 8 / 4 2.2 = 68.727  
/ * 7 + 8 4 2 = 42.000
```

```
Test 2: Test mit falscher Grammatik:  
2 + 3 = 2.000  
6 / 3 = 6.000  
7 * 8 + (4 / 2) = 7.000
```

```
Test 3: Test mit Variablen:  
Error: Error parsing `Expression`  
Error: Error parsing `Expression`  
Error: Error parsing `Expression`
```

```
Test 4: Test mit Klammern:  
Error: Error parsing `Expression`
```

```
Test 5: Test mit Division durch 0:  
Error: Error trying to divide by zero!  
Error: Error trying to divide by zero!
```

```
Test 6: Test mit leerem String:  
Error: Error parsing `Expression`  
Error: Error parsing `Expression`
```

Beispiel 3:

Lösungsidee:

Dieses Beispiel benutzt die gleiche Grammatik wie Beispiel 1, jedoch sollten wir nun mit Variablen, also Platzhaltern für Zahlen rechnen können. Dazu erstellt man sich eine Map, mit welcher man sich den Variablen-Namen als Schlüssel und die dazugehörige Zahl als Value einspeichert. Diese Map ist Teil der Klasse. Wichtig ist auch noch, dass man die Map als Ungeordnete Map implementiert, da die Variablen sonst nicht an der richtigen Stelle sitzen und nicht richtig abgerufen werden können.

Um herauszufinden, ob ein Symbol eine Variable ist, habe ich eine zusätzliche Funktion gebaut. Diese überprüft, ob das Symbol nichts der anderen Möglichen Symbolen, also keine Binary-Operation, keine Zahl, und keine Klammer ist. Wenn das alles zutrifft, muss es sich nur um eine Variable handeln, und die Funktion retourniert true.

Wir müssen sonst nur mehr die Parse-Faktor-Funktion verändern, da wir hier immer auf die Zahlen und Operations-Symbole zugreifen wollen. Wir überprüfen nun also, ob es sich bei dem aktuellen Symbol um eine Variable handelt. Wenn ja, dann müssen wir uns die Bezeichnung dieser Variable speichern, um sie anschließend in unserer Map zu suchen. Das machen wir mit einer get-Funktion aus dem pfc_scanner-File und speichern das Ganze als string. Wir benötigen hier das Attribut des aktuellen Symbols.

Wenn wir die Variable innerhalb der Map finden können, speichern wir uns die mit dem Schlüssel allokierte Zahl und rücken wieder um eine Stelle nach vorne. Sonst müssen wir hier nichts verändern und das Beispiel sollte so funktionieren.

Testfälle:

Test 1: Test mit mehreren Rechnungen:

```
a * 2 = 8.000  
b / 3 = 2.333  
a + var = 6.000
```

Test 2: Test mit negativen Zahlen:

```
a * 2 = -8.000  
b / 3 = -2.333  
a + var = -6.000
```

Test 3: Test mit falscher Grammatik:

```
Error: Error parsing `Expression`  
Error: Error parsing `Expression`
```

Test 4: Test mit leeren strings:

```
Error: bad variant access  
Error: bad variant access  
Error: bad variant access
```

Test 5: Test mit Division durch 0:

```
Error: Error trying to divide by zero!  
Error: Error trying to divide by zero!  
Error: Error trying to divide by zero!
```