Beispiel 1: Ausdrucksbaum

Lösungsidee:

Ziel ist es einen Parser zu entwickeln, welcher folgende Grammatik unterstützt:

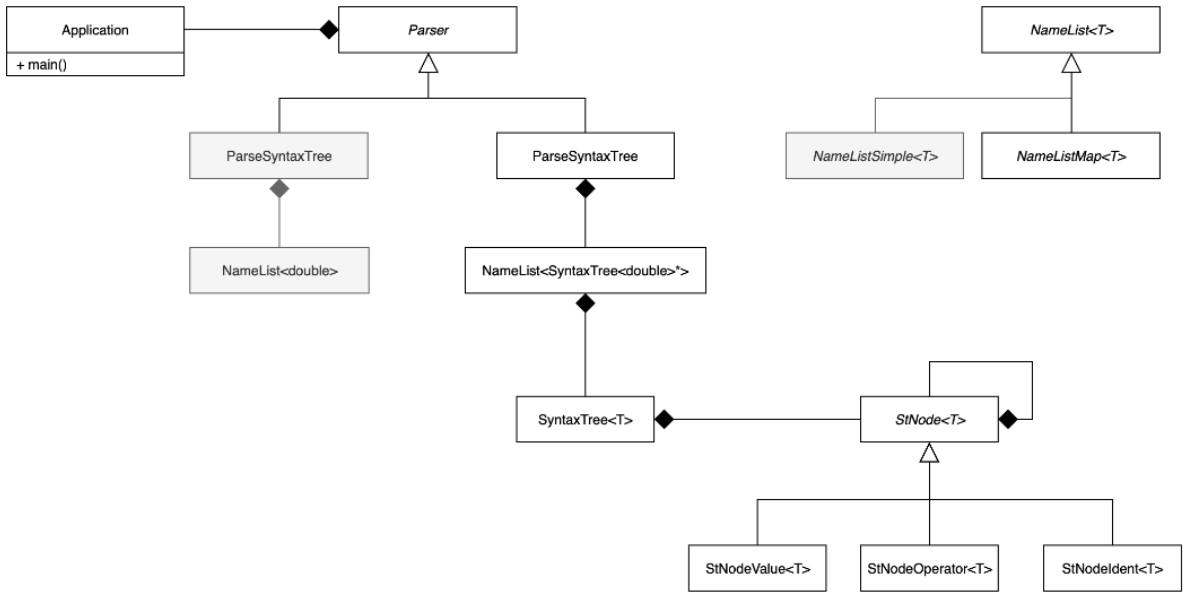
```
Programm  = { Ausgabe | Zuweisung } .
Ausgabe   = „print“ „(“ Ausdruck „)“ „;“ .
Zuweisung = „set“ „(“ Identifier „,“ Ausdruck „)“ „;“ .

Ausdruck  = Term { AddOp Term } .
Term      = Faktor { MultOp Faktor } .
Faktor    = [ AddOp ] UFaktor .
```

```
UFaktor   = Monom | KAusdruck .
Monom     = WMonom [ Exponent ] .
KAusdruck = „(“ Ausdruck „)“ .
WMonom    = Identifier | Real .
Exponent  = „^“ [ AddOp ] Real .
AddOp     = „+“ | „-“ .
MultOp    = „*“ | „/“ .
```

Ein Programm besteht aus einer beliebigen Anzahl von Anweisungen, die entweder eine Ausgabe oder eine Zuweisung sein können. Eine Ausgabe besteht aus dem Schlüsselwort "print", gefolgt von einem Ausdruck in Klammern und einem Semikolon. Eine Zuweisung besteht aus dem Schlüsselwort "set" gefolgt von einem umklammerten Paar aus Identifier und Ausdruck. Der übergebene Ausdruck soll also unter dem angegeben Identifier gespeichert werden. Ansonsten ist die Grammatik ähnlich aufgebaut wie jene aus der letzten Übung, nur dass, durch einige weitere Regeln bspw. nun auch das Rechnen mit Exponenten möglich sein soll.

Grundsätzlich soll das Programm folgendem Aufbau folgen:



Die Parser-Klasse dient lediglich dazu Vorgaben daran zu stellen, welche Methoden die Klasse ParseSyntaxTree enthalten muss. SyntaxTree soll eine Baumstruktur bestehend aus Nodes ermöglichen. Nodes wiederum können entweder Werte (StNodeValue), Operatoren (StNodeOperator) oder Variablen bzw. Identifier (StNodeIdent) beinhalten.

Zudem soll es möglich sein, ganze Ausdrücke in einer NameList, also jene Liste, die die Werte für Identifier beinhaltet, zu speichern. Über den Aufbau eines Ausdrucksbaumes, welcher verschiedene Arten von Knoten beinhalten kann (Knoten für Werte, Operatoren und auch Identifier) sollen die Ausdrücke aufgebaut werden indem aus den Werten des arithmetischen Ausdrucks während des Parsens Knoten aufgebaut werden, welche dann im späteren Verlauf zu einem Ausdruckbaum zusammengebaut werden. Dieser kann dann evaluiert und ausgegeben werden.

Für den Aufbau des Ausdrucksbaumes bietet sich das Verwenden eines Stacks an. Man erstellt 2 Stacks, einen für die Operanden und einen für die Operatoren des Ausdrucks. Nun geht man den Ausdruck mithilfe des Scanners Stück für Stück durch und fügt dabei Zahlen zum Operanden-Stack und Operatoren zum Operatoren-Stack hinzu. Mit Berücksichtigung einiger weiterer Faktoren erhält man nun einen Stack, welcher in so einer Reihenfolge aufgebaut ist, dass man daraus einen Ausdrucksbaum erstellen kann, welcher dann evaluiert wird.

Neben der normalen Ausgabe der Ausdrücke bzw. des Ausdrucksbaumes ist auch eine 2-dimensionale Ausgabe des Baumes möglich (sowohl in waagrechter, als auch senkrechter Darstellung). Für die waagrechte Variante geht man hierfür den Baum von rechts nach links durch und gibt die Werte der Elemente Zeile für Zeile aus. Hierbei muss man natürlich über variable Abstände die Ausgabe übersichtlich gestalten. Bei der vertikalen Variante ist es wichtig die Höhe des Baumes in Erfahrung zu bringen, da man diesen dabei Ebene für Ebene durchgeht und ausgibt. Nach dem Übergehen in eine nächste Ebene sollte man auch den Abstand zwischen einzelnen Werten verringern, da dieser anfangs ja relative groß sein sollte.

Bemerkung: Mir war es leider zeitlich nicht mehr möglich eine anständige Lösung für das Parsen eines Audrucks zu entwickeln, unter anderem da der Scanner die keywords als identifier wahrgenommen hat (konnte meinen Fehler nicht entdecken) und somit ein Überprüfen schwierig war.

Tests:

Präfix, Infix und Postfix Ausgabe eines simplen Ausdrucks:

```
testing simple trees:
Print tree in pre-order notation: + 17 4
Print tree in in-order notation: 17 + 4
Print tree in post-order notation: 17 4 +
```

2-dimensionale Ausgabe eines simplen Ausdrucksbaumes:

```
Print tree in 2D (horizontally):
                    4
          +
                    17


Print tree in 2D (vertically):

 +
17 4
```

Präfix, Infix und Postfix Ausgabe eines komplexeren Ausdrucks:

```
testing more complex trees:
Print tree in pre-order notation: + * + 2 4 3 / 5 2
Print tree in in-order notation: 2 + 4 * 3 + 5 / 2
Print tree in post-order notation: 2 4 + 3 * 5 2 / +
```

2-dimensionale Ausgabe eines komplexeren Ausdrucksbaumes:

```
Print tree in 2D (horizontally):
                              2
                    /
                              5
          +
                              3
                    *
                                        4
                              +
                                        2


Print tree in 2D (vertically):

   +
  * /
 + 3 5 2
2 4
```

Simple Verwendung der NameList:

Für x wird folgender Wert festgelegt:

```
StNodeValue<double>* x_value = new StNodeValue<double>(3);
```

```
testing usage of namelist:
Addition:
Print tree in pre-order notation: + 17 x
Print tree in in-order notation: 17 + x
Print tree in post-order notation: 17 x +
Result: 20
```

Etwas komplexere Verwendung der NameList:

Für x wird folgender Wert festgelegt:

```
StNodeValue<double>* x_value = new StNodeValue<double>((9-2) * 2.5);
```

```
testing more complex of namelist:
Addition:
Print tree in pre-order notation: + 17 x
Print tree in in-order notation: 17 + x
Print tree in post-order notation: 17 x +
Result: 34.5
```

Mögliche Testfälle für die Programm-Grammatik wären:

Folgende Eingabe sollte zu einem Fehler führen, da das Semikolon am Ende fehlt:

```
set(x, 3)
```

Folgende Eingabe sollte den Wert 8 für x speichern:

```
set(x, 4*2);
```

Nach dem Abspeichern des Wertes für x sollte folgende Print-Anweisung das Ergebnis 20 liefern:

```
print((48-x)/2);
```

Hat man für den Wert y zuvor 0 festgelegt, sollte folgende Anweisung eine DivByZeroException hervorrufen:

```
print(14/y);
```

Wurde für die Variable z in der NameList zuvor kein Eintrag angelegt, führt diese Anweisung zu einem Fehler, da das Zeichen z nicht bekannt ist:

```
print(3 - z);
```

Folgender Ausdruck würde auch zu einem Fehler führen, da zu Beginn kein gültiges Schlüsselwort (in diesem Fall gar keins) ist:

```
(3*4 + 8/2)
```

Diese Anweisung sollte das Ergebnis 9 liefern, für den Fall, dass für x der Wert 2 festgelegt ist:

```
print(3^2)
```