Algorithmen und Komplexität
Laborprojekt (C++)

Kurs TINF22ITA

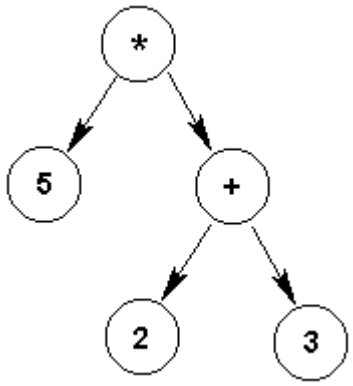
Bearbeitungszeit: 26.06.2023 – 11.07.2023

Hinweise zur Abgabe des Projekts:

- spätestens am Dienstag, den 11.07.2023
- verpacken Sie das Projektverzeichnis mit allen Dateien und Unterverzeichnissen in eine ZIP-Datei, die als Namen die Nachnamen der Gruppenmitglieder trägt, z.B. „Labor - Meier, Müller, Wagner.zip“.
- schicken Sie die ZIP-Datei per E-Mail an fritzsch@lehre.dhbw-stuttgart.de
  Im Text der Mail sind nochmals die Namen aller Gruppenmitglieder gelistet
- nur ein Gruppenmitglied schickt das Projekt ein, bei mehreren Abgaben einer Gruppe wird nur die letzte gewertet

Thema des Projekts: Arithmetische Bäume

Das Programmierprojekt beschäftigt sich mit der Darstellung arithmetischer Ausdrücke als Binärbaum. Beispielsweise lässt sich der Term „5 * (2 + 3)“ als Binärbaum wie folgt darstellen:



Ein arithmetischer Binärbaum hat also eine Wurzel (das „x“) und zwei Arten von Knoten: **Rechenoperatoren** mit jeweils zwei Unterbäumen. Diese Unterbäume repräsentieren die Operanden. **Zahlen** werden durch Blätter repräsentiert. Ein arithmetischer Baum kann auf verschiedene Arten dargestellt werden:

- Infix-Notation: Operatoren stehen zwischen ihren Operanden: `(5 * (2 + 3))`
- Prefix-Notation: Operatoren stehen vor ihren Operanden: `* 5 + 2 3`
- Postfix-Notation: Operatoren stehen hinter ihren Operanden: `5 2 3 + *`

Bei der Infix-Darstellung soll der Ausdruck vollständig geklammert sein:

5 * 2 + 3 oder 5 * (2 + 3) genügen nicht, die Ausdrücke müssen folgende Form haben: ((5 * 2) + 3) bzw. (5 * (2 + 3)).
Bei Pre- und Postfix-Darstellung gibt es keine Klammern.
Im Projekt geht es darum, arithmetische Ausdrücke in allen drei Darstellungen einlesen zu können, daraus den arithmetischen Baum aufzubauen und diesen auszuwerten:

Welches Ergebnis hat der Ausdruck? -> im Beispiel: 25
Wie viele Knoten hat der Baum? -> im Beispiel: 5
Wie tief ist er? -> im Beispiel 3

Es steht einProjektvorlage für MS Visual C++ 2019 (**ABaumVCpp**) zur Verfügung. Als Hilfe ist eine Klassenstruktur vorgegeben und einige der Klassen sind auch schon teilweise implementiert. Diese Grundlage können Sie für Ihre Umsetzung benutzen. Für die Ausgabe des Baumes ist eine Konsolenausgabe (ASCII-Grafik) ausreichend.

Beschreibung der Klassenstruktur

Ein Knoten im Baum wird durch die abstrakte Klasse **Token** und deren Unterklassen dargestellt. Abstrakt bedeutet, dass in der Klasse nicht alle Methoden implementiert sind. Einige sind als *virtual* deklariert und müssen dann in den erbenden Klassen implementiert werden.

Die Klasse **Token**:

- stellt sicher, dass jeder Baumknoten bestimmte Methoden bietet
- *int eval()* soll den Baum zu einem Zahlenwert auswerten
- *String pre-, in-* und *postfix()* sollen eine String-Darstellung der Form
  " * 5 + 2 3 " bzw. " (5 * (2 + 3)) " bzw. " 5 2 3 + * " zurückgeben
- *int nodes()* soll die Anzahl der Knoten des Baums bzw. Unterbaums zurückgeben
- *int depth()* soll die Tiefe des Baums bzw. Unterbaums zurückgeben

Die Klasse **Num**:

- **Num** stellt einen numerischen Wert dar, daher muss die Klasse einen int-Wert speichern
- Es soll neben dem leeren Standardkonstruktor auch einen Konstruktor geben, der einen int erwartet, das int-Attribut damit initialisiert und den Knotentyp type auf 'n' für numerisch setzt.
- Die Auswertung *int eval()* soll den numerischen Wert des Knotens zurückgeben.
- *pre-, in-* und *postfix()* sollen die Zahl als String zurückgeben.
- Da **Num** keine Unterbäume hat, sind sowohl die Anzahl der Knoten (*nodes()*) als auch die Tiefe (*depth()*) gleich 1. Überlegen Sie, ob Sie die beiden Methoden dann überhaupt implementieren müssen.

Die Klasse **Op**:

- *eval()* soll den Operator (type kann '+', '–', '*' oder '/' sein) auf die rekursive Auswertung des linken und rechten Unterbaums anwenden und das Ergebnis zurückgeben
- *pre-, in-* und *postfix()* sind ebenfalls rekursiv: der Operator (type) soll vor, zwischen oder nach den entsprechenden Darstellungen der Unterbäume stehen. Bei der Infix-Darstellung sind die Klammern nicht zu vergessen.
- *nodes()* soll die Anzahl der Knoten zurückgeben. Berücksichtigen Sie die Unterbäume (Rekursion).
- *depth()* soll rekursiv die Tiefe des Baums zurückgeben.

Testen können Sie die Funktionen, indem Sie in der Methode *evaluate()* der Klasse **Evaluator** die auskommentierte Testbaum-Zuweisung nutzen und dann die *main()*-Methode mit einem beliebigen String-Array, z.B. {"1"} aufrufen. Die Ergebnisse sollten mit denen in den Kommentaren übereinstimmen.

Die Klasse **Tokenizer**:

Ein Tokenizer zerlegt einen String in Teile, so genannte Tokens. Die benötigten Funktionen eines Tokens überschneiden sich mit denen eines Baumknotens (von beiden muss man den Typ abfragen können), daher leisten Token und seine Unterklassen beides. Es gibt drei Arten von Tokens: Zahlen (**Num**), Operatoren (**Op**) und Klammern (**Bracket**). Zahlen haben einen Wert, Operatoren gibt es in vier Typen ('+', '–', '*' und '/'), Klammern in zwei ('(' und ')').
Die Methode *tokenize()* der Klasse **Tokenizer** soll den String *src* Zeichen für Zeichen durchgehen und daraus eine Folge von Tokens erzeugen. Da die Länge der Folge nicht von vornherein feststeht, ist ein Array keine Option. Stattdessen wird ein *Vector* benutzt – die Länge eines Vectors ist, wie die einer verketteten Liste, veränderlich. Machen Sie sich in der C++ STL Klassenbibliothek mit der Klasse *Vector* vertraut. In unserem Fall stellen wir sicher, dass der *Vector* (der allgemein Elemente des Typs *Object* enthält) nur Tokens enthält, d.h. wir definieren ein Objekt vom Typ vector<Token*>.

Vorgehensweise: Wenn das aktuelle Zeichen im String eine Klammer ist, fügen Sie dem *Vector* mittels dessen Methode *add(...)* ein neues Bracket-Objekt mit dem korrekten Typ hinzu. Bei einem Operator genauso, auch hier müssen Sie dem Konstruktor den korrekten Typ übergeben. Bei einer Ziffer zwischen '0' und '9' müssen Sie so lange weiterlesen, wie die nächsten Zeichen auch Ziffern sind. Aus der gelesenen Zahl (Sie müssen überlegen, wie Sie die einzelnen Ziffern zu einem int zusammensetzen) erzeugen Sie ein neues **Num**-Objekt und fügen dieses dem Vector hinzu. Jedes andere Zeichen ignorieren Sie einfach. Achten Sie darauf, keine Zeichen zu überspringen – der Tokenizer sollte sowohl mit "`( 42 - 23 )`", als auch mit "`(42-23)`", "`+ 1 * 2 3`" und "`1 2 3*+`" klarkommen.

Ihren Tokenizer können Sie testen, indem Sie ein Objekt der Klasse **Tokenizer** erzeugen. Darauf rufen Sie dann die Methode *tokenize()* auf – Sie erhalten einen vector<Token*>.

Die Klasse **Evaluator**:

Die Aufgabe des Parsers ist es, aus dem Vector von Tokens, den der Tokenizer liefert, einen Baum aufzubauen. Da der arithmetische Ausdruck in drei verschiedenen Notationen vorliegen kann, müssen Sie allerdings drei verschiedene Parser schreiben. Die Methode *parse(…)* in der Klasse **Evaluator** ruft je nach Notation den richtigen Parser auf und erzeugt vorher einen *Iterator* über den Token-Vector. Ein *Iterator* ist dazu da, Elemente in z.B. einem *Vector* der Reihe nach zu durchlaufen. Machen Sie sich hierzu in der C++ STL Klassenbibliothek mit der Klasse *Iterator* vertraut.

Nun sollen bei den drei Parsern die zu implementierenden Stelle n mit Inhalt gefüllt werden.

Prefix-Parser: parsePrefix()

In der Schleife gibt es für das gelesene Element zwei Möglichkeiten. Wenn es sich um eine Zahl handelt (*type()* liefert 'n'), geben wir das Element als Ergebnis der Methode zurück. Wenn nicht, handelt es sich um einen Operator. Da der Ausdruck in Prefix-Notation vorliegt, folgen dem Operator die beiden Operanden. Diese sind allerdings auch wieder arithmetische Ausdrücke in Prefix-Notation: Beim Ausdruck "* + 2 + 3 4 5" ist der erste Operand des Multiplikationsoperators der Ausdruck "+ 2 + 3 4", der zweite die Zahl 5. Also müssen Sie rekursiv erst den linken Operand parsen (das Iterator-Objekt i übergeben Sie dabei einfach unverändert) und das Ergebnis zwischenspeichern, dann den rechten. Mit diesen Daten können Sie dann das Ergebnis der Methode zurückgeben: ein neuer Op-Knoten des richtigen Typs mit den gerade rekursiv geparsten Operanden als linken und rechten Unterbaum.

Postfix-Parser: parsePostfix()

Bei der Postfix-Notation ist ein Operator immer hinter seinen beiden Operanden. Während der Token-Vector durchlaufen wird, dürfen wir die gelesenen Elemente also nicht vergessen haben, wenn wir zu einem Operator kommen. Dafür nutzen wir einen *Stack* (Zur Erinnerung: neue Elemente legen Sie oben drauf, und wenn Sie ein Element vom Stapel nehmen, ist es immer das Element, welches zuletzt auf den Stapel gelegt wurde). Machen Sie sich hierzu in der C++ STL Klassenbibliothek mit der Klasse *Stack* vertraut. Ein leerer *Stack*, der Tokens aufnimmt, ist unter dem Namen *s* in der Methode schon vorhanden.

In der Schleife gibt es für das gelesene Element wieder zwei Möglichkeiten, allerdings reagieren wir jetzt anders. Wenn es sich um eine Zahl handelt, legen wir diese einfach auf den Stapel. Wenn nicht, müssen Sie die beiden Operanden vom Stapel nehmen und zwischenspeichern. Erzeugen Sie dann einen neuen Op-Knoten des richtigen Typs mit den gerade vom Stapel genommenen Operanden als linken und rechten Unterbaum und legen Sie diesen Op-Knoten wieder auf den Stapel. Nach der while-Schleife sollte sich bei einem korrekten Postfix-Ausdruck jetzt genau ein Element auf dem Stapel befinden – geben Sie dieses als Ergebnis der Methode zurück.

Infix-Parser: parseInfix()

Bei Infix-Ausdrücken kommen zu den möglichen Elementen im Token-Vector auch noch Klammern. Da die Ausdrücke vollständig geklammert sind, markiert eine schließende Klammer ')' immer das Ende eines Teilausdrucks – das machen wir uns zunutze. Wenn das in der Schleife gelesene Element keine schließende Klammer ist, legen wir es einfach auf den Stapel. Wenn doch, müssen wir den Stapel abarbeiten: die drei obersten Elemente sind der Operator und seine beiden Operanden – diese drei Elemente müssen Sie in sinnvoll benannten Variablen zwischenspeichern (überlegen Sie, in welcher Reihenfolge die Elemente auf dem Stapel liegen).

Nehmen Sie dann noch ein Element vom Stapel, ohne es zwischenzuspeichern – das ist die öffnende Klammer.
Erzeugen Sie dann einen neuen Op-Knoten des richtigen Typs mit den gerade vom Stapel genommenen Operanden als linken und rechten Unterbaum und legen Sie diesen Op-Knoten wieder auf den Stapel. Nach der while-Schleife sollte sich bei einem korrekten Infix-Ausdruck jetzt genau ein Element auf dem Stapel befinden – geben Sie dieses als Ergebnis der Methode zurück.

Machen Sie sich die Funktionsweise der drei Parser zuerst mit Papier und Stift klar.

Testen

Zum Testen Ihres Projekts nutzen Sie die *main()*-Funktion/Methode. Dazu macht es nichts, wenn Sie noch nicht alle drei Parser fertig implementiert haben. Übergeben Sie als Parameter einfach einen Ausdruck in einer Notation, deren Parser schon fertig ist.

Das Eingabeformat der *main()*-Funktion/Methode sieht folgendermaßen aus:
Die Elemente des String-Arrays args bzw. argv werden zu einem String zusammengefügt. Dessen erstes Zeichen bestimmt die Notation: '<' für Prefix, '>' für Postfix, '|' oder etwas anderes für Infix. Der restliche String (bzw. der ganze, wenn das erste Zeichen nicht <, > oder | war) enthält dann einen arithmetischen Ausdruck in der entsprechenden Notation. Leerzeichen sind dabei nur nötig, um zwei Zahlen voneinander zu trennen.

Beispiele:

```
{"<", "/+10*4 5-/84 12 1"}
{"(  (10 + (4*5)  ) / ((84/12)-1))"}
{">10 4 5*+84 12 / 1 -/"}
```

Alle Ausdrücke sollten den gleichen Baum mit 11 Knoten, Tiefe 4 und Ergebnis 5 liefern.
Probieren Sie in der Methode *evaluate()* beide Baumdarstellungen der Klasse **Vis** aus.

Dokumentation

Vervollständigen Sie die Kommentierung der Klassen **Evaluator**, **Tokenizer**, **Token**, **Op** und **Num**. Jede Klasse sollte einen Kommentar mit Autor- und Versionsangabe haben, jeder Konstruktor und jede Methode einen Kommentar, der ihre Aufgabe, Parameter und Rückgabewerte beschreibt.

Und nun viel Erfolg!