Technische Universität Darmstadt
Fachbereich Informatik
Prof. Dr. Johannes Fürnkranz

Allgemeine Informatik 2 im SS 2007

Programmierprojekt

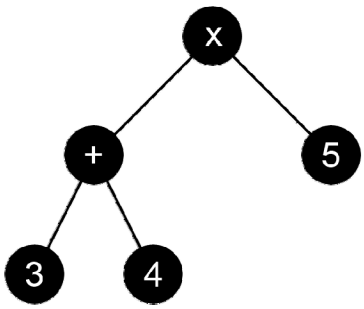
Bearbeitungszeit: 04.06. bis 13.07.2007

Die Formalitäten vorab (WICHTIG!!!):

- Beachten Sie alle Hinweise zum Programmierprojekt auf der Seite http://www.ke.informatik.tu-darmstadt.de/lehre/ss07/ai2/uebungen.php!
- Tragen Sie in die Readme-Datei des BlueJ-Projekts (das Symbol links oben auf der BlueJ-Arbeitsfläche) Namen und Matrikelnummern aller Gruppenmitglieder ein!
- Abgabe des Projekts:
  - spätestens am 13.07.2007.
  - Verpacken Sie das Projektverzeichnis mit allen Dateien und Unterverzeichnissen in eine ZIP-Datei, die als Namen die Matrikelnummer eines Gruppenmitglieds trägt, also z.B. „1234567.zip“.
  - Im Terminal / in der Shell geht das so: **`zip -r 1234567.zip propro`**
  - Schicken Sie die ZIP-Datei per E-Mail an allginf@gmx.de.
  - Der Betreff soll dabei gleich dem Namen der ZIP-Datei sein.
  - Im Text der Mail sollen noch mal die Namen und Matrikelnummern aller Gruppenmitglieder stehen.
  - Nur ein Gruppenmitglied schickt das Projekt ein!
  - Bei mehreren Abgaben einer Gruppe wird nur die letzte gewertet.

Arithmetische Bäume

Das Programmierprojekt beschäftigt sich mit arithmetischen Bäumen. Der Term „(3 + 4) * 5“ sieht als Baum folgendermaßen aus:



Ein arithmetischer Baum hat also eine Wurzel (das „x“) und zwei Arten von Knoten: Rechenoperatoren mit jeweils zwei Unterbäumen als Operanden und Zahlen ohne Unterbäume. Ein AB kann auf verschiedene Arten dargestellt werden:

- Infix: Operatoren stehen zwischen ihren Operanden: **`((3 + 4) * 5)`**.
- Prefix: Operatoren stehen vor ihren Operanden: **`* + 3 4 5`**.
- Postfix: Operatoren stehen hinter ihren Operanden: **`3 4 + 5 *`**.

Bei der Infix-Darstellung muss der Ausdruck **vollständig geklammert** sein, um eindeutig zu ein: **`3 + 4 * 5`** oder **`(3 + 4) * 5`** reichen nicht aus. 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: 35] Wie viele Knoten hat der Baum? [5] Wie tief ist er? [3]). Als Hilfe ist die Klassenstruktur schon vorgegeben und einige der Klassen sind auch schon vollständig, z.B. die grafische Darstellung des Baums in der Klasse **`Vis`**.

Teil 1: Vorbereitung

Laden Sie sich von der Übungs-Webseite auch die Datei **propro.zip** herunter und entpacken Sie sie (**`unzip propro.zip`** im Terminal). Starten Sie dann BlueJ und öffnen Sie das Projekt **propro**. Füllen Sie zuerst wie oben beschrieben die Readme aus (Doppelklick auf das Symbol links oben)!

Teil 2: Baumknoten

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 **`abstract`** deklariert und müssen dann in erbenden Klassen implementiert werden. Näheres dazu in der Vorlesung.

Die Klassen **`Token`** und **`Bracket`** sind (bis auf die Kommentierung) schon vollständig, wobei **`Bracket`** erst in Teil 3 interessant wird.

`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 **`"* + 3 4 5"`** bzw. **`"((3 + 4) * 5)"`** bzw. **`"3 4 + 5 *"`** zurückgeben.
- **`int nodes()`** soll die Anzahl der Knoten des Baums bzw. Unterbaums zurückgeben.
- **`int depth()`** soll die Tiefe des Baums bzw. Unterbaums zurückgeben.

Vervollständigen Sie 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'`** setzt.
- Die Auswertung **`int eval()`** soll den numerischen Wert des Knotens zurückgeben.
- **`pre`**-, **`in`**- und **`postfix()`** sollen die Zahl als String zurückgeben: **`"" + <Zahl>`**.
- Da **`Num`** keine Unterbäume hat, sind sowohl die Anzahl der Knoten (**`nodes()`**) als auch die Tiefe (**`depth()`**) gleich 1 – überlegen Sie also, ob Sie die beiden Methoden überhaupt implementieren müssen.

Vervollständigen Sie 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 die Klammern nicht vergessen!
- **`nodes()`** soll die Anzahl der Knoten zurückgeben. Berücksichtigen Sie die Unterbäume (Rekursion)!
- **`depth()`** soll rekursiv die Tiefe des Baums zurückgeben. **Tipp:** die Maximumsfunktion lautet **`Math.max(a, b)`**.

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.

Teil 3: 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 vorneherein feststeht, ist ein Array keine Option. Stattdessen wird ein **`Vector`** benutzt – die Länge eines Vectors ist nämlich wie die einer verketteten Liste veränderlich. Machen Sie sich in der **Java-API** mit der Klasse **`Vector`** vertraut! In unserem Fall stellen wir sicher, dass der **`Vector`** (der allgemein Elemente des Typs **`Object`** enthält) nur **`Token`**s enthält, d.h. wir definieren ein Objekt vom Typ **`Vector<Token>`** (diese so genannten „**Generics**" funktionieren erst ab Java 1.5). Näheres auch in der Vorlesung.

Die Vorgehensweise: wenn das aktuelle Zeichen im String eine Klammer ist, fügen Sie dem **`Vector`** mittels dessen Methode **`add(...)`** ein neues **`Bracket`**-Objekt hinzu (mit dem korrekten Typ). 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 muss sowohl mit **`"( 42 - 23 )"`** als auch mit **`"(42-23)"`**, **`"+ 1 * 2 3"`** und **`"1 2 3*+"`** klarkommen.

Ihren Tokenizer können Sie testen, indem Sie in BlueJ ein Objekt der Klasse **`Tokenizer`** erzeugen. Darauf rufen Sie dann die Methode **`tokenize()`** auf – Sie erhalten einen **`Vector<Token>`**. Mittels „**Inspect**" können Sie diesen und das enthaltene Array **elementData** untersuchen.

Teil 4: Parser

Die Aufgabe des Parsers ist es, aus dem **`Vector`** von **`Token`**s, 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 in der **Java-API** mit der Klasse **`Iterator`** vertraut!
Auch hierzu gibt es nähere Informationen in der Vorlesung.

Der Iterator bietet für uns zwei wichtige Methoden:

- **`boolean hasNext()`**: **`true`**, wenn es noch weitere Elemente im Token-Vector gibt.
- **`Token next()`**: liefert den jeweils nächsten Token im Vector (zu Beginn also den ersten).

Nutzen Sie in allen drei Parser-Methoden die beiden **`Iterator`**-Methoden in einer **`while`**-Schleife, um den Token-Vector zu durchlaufen – ähnlich zu diesem Code, der ein Array **`a`** durchläuft:

```
int i = 0;
while (i < a.length) {
  int element = a[i];
  i++;
  // Und jetzt irgendetwas mit dem Element tun... }
```

Die Initialisierung "i = 0" lassen Sie allerdings weg, starten Sie einfach da, wo der Iterator gerade steht. Die Schleifenbedingung muss im Parser natürlich überprüfen, ob es noch weitere Elemente gibt, und die Funktion des Zugriffs auf ein Element und das Weiterrücken des Zählers übernimmt die Methode **`next()`**. Das Element müssen Sie analog zum Beispiel zwischenspeichern.

Jetzt sind wir bereit, bei den drei Parsern die Stelle „**`// Und jetzt irgendetwas mit dem Element tun...`**" mit Inhalt zu füllen!

Prefix-Parser: `private static Token parsePrefix(Iterator<Token> i)`

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: `private static Token parsePostfix(Iterator<Token> i)`

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`**. Ein Stack funktioniert wie ein Stapel: 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 in der **Java-API** mit der Klasse **`Stack`** vertraut! Ein leerer **`Stack`**, der **`Token`**s aufnimmt, ist unter dem Namen **`s`** in der Methode schon vorhanden.

Die für uns wichtigen Methoden:

- **`push(Token e)`**: legt ein Element auf den Stapel.
- **`Token pop()`**: nimmt das oberste Element vom Stapel und gibt es zurück.

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: `private static Token parseInfix(Iterator<Token> i)`

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!

Teil 5: Testen

Zum Testen Ihres Projekts nutzen Sie die **`main`**-Methode der Klasse **`Evaluator`**. Dazu macht es nichts, wenn Sie noch nicht alle drei Parser fertig haben, übergeben Sie als Parameter einfach einen Ausdruck in einer Notation, deren Parser schon fertig ist. Vergessen Sie nicht, den Testbaum wieder auszukommentieren!

Das Eingabeformat der **`main`**-Methode sieht folgendermaßen aus:
Die Elemente des String-Arrays **`args`** werden zu einem String zusammengefügt. Dessen erstes Zeichen bestimmt die Notation: **`'<'`** ➔ Prefix, **`'>'`** ➔ Postfix, **`'|'`** oder etwas anderes ➔ 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(…)`** ruhig beide Baumdarstellungen der Klasse **`Vis`** aus!

Teil 6: JavaDoc

Vervollständigen Sie die JavaDoc-Kommentierung der Klassen **`Evaluator`**, **`Tokenizer`**, **`Token`**, **`Op`** und **`Num`**.

Jede Klasse muss einen Kommentar mit Autor- und Versionsangabe haben; jeder Konstruktor und jede Methode muss einen Kommentar haben, der die Funktion, Parameter und Rückgabewerte beschreibt!

Viel Erfolg!