



SWE3_DEUTZ_UE07

Übungszettel 07

AUFGABE 1: AUSDRUCKSBAUM

LÖSUNGSIDEE

PARSER

Der zu implementierende Parser soll in der Lage sein ein einfaches Programm für das Speichern und Schreiben von Ausdrücken auszuwerten und auszuführen. Hierbei sind die beiden Schlüsselwörter *set* und *print* entscheidend und werden demnach entsprechend behandelt. Der Parser folgt grundsätzlich der nachfolgenden Grammatik:

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	= „*“ „/“.
Real	= Grammatik in Scanner eingebaut

Für das Einlesen der einzelnen Zeichen wird der Scanner aus dem Namensraum *pfc* verwendet. Zu Beginn des Parsvorganges werden die beiden Schlüsselwörter *set* und *print* beim Scanner registriert und der entsprechende Stream gesetzt.

Die Klasse bietet zwei Möglichkeiten für das Parsen eines Programms: mithilfe eines offenen Streams oder eines Dateinamens. Sollte ein Dateiname übergeben werden, wird ein Stream geöffnet und mit diesem weitergearbeitet.

Wie schon bei Übungszettel 06 werden mithilfe der *is_tb* Funktionen nach unten absteigend die Regeln abgearbeitet und überprüft, ob sie erfüllt wurde. Da dies in der Lösungsidee schon erläutert wurde, verweise ich darauf.

Im Unterschied zu Übungszettel 06 wird in den *parse* Funktionen nun ein Baum zusammengebaut, welcher anschließend ausgewertet und gespeichert bzw. ausgegeben wird, je nach dem aktuellen Schlüsselwort. Das Zusammenbauen des Baumes erkläre ich anhand des Beispiel Ausdruck:

Es wird ein Zeiger auf einen Knoten erstellt und im Falle eines Ausdruckes der Term ausgewertet, aus dem der Ausdruck mindestens bestehen muss. Ist der Term

ausgewertet wird überprüft, ob noch eine Addition oder eine Subtraktion vorhanden ist. Solange dies der Fall ist, wird ein Knoten für die AddOp erstellt und der zweite Term ausgewertet und wieder ein Zeiger für den „Termbaum“ erstellt. Am Ende wird der Baum neu zusammengestellt, sodass der Operatorknoten zum Wurzelknoten und die beiden Termbäume dessen Kinder werden. Dieses Prinzip wird überall angewandt, nur bei Bäumen von Exponentoperatoren gibt es den Unterschied, dass diese nur einen rechten Teilbaum besitzen bevor der Zeiger retourniert wird, der linke Teilbaum wird in der Ebene darüber hinzugesteckt.

NAMENSLISTE

Die Namensliste dient zur Speicherung der Variablennamen sowie deren Werte. Die Basisklasse *NameList* wurde generisch als Interface implementiert, was bedeutet, es gibt weder einen Konstruktor oder Datenkomponenten und die Methoden wurden nur deklariert und nicht definiert. Sie besteht also rein aus „pure virtual functions“. Die Namensliste bietet Methoden zum Registrieren und Abfragen von Variablen sowie einer Methode zum Ausgeben einer vorhandenen Namensliste. Im folgenden Screenshot ist zur besseren Veranschaulichung die finale Version des Interface zu sehen:

```
template<typename T> class NameList {
public:
    virtual ~NameList(){};

    virtual T lookup_var(std::string const& _name_) = 0;
    virtual void register_var(std::string const& _name_, T const& _value_) = 0;
    virtual std::ostream& print(std::ostream& _out_ = std::cout) const = 0;
};
```

Es existiert eine generische Spezialisierung der Basisklasse, welche das Speichern der Variablen mithilfe einer *std::map* realisiert und ebenfalls später für den Parser verwendet wird. Eine map speichert nach dem Prinzip von *Key-Value-Pairs* und bietet somit eine einfache Möglichkeit für das Einfügen und Abfragen neuer bzw. existierender Paare.

Anmerkung: Aufgrund der in der Übung besprochenen Implementierungsweise der Namensliste funktioniert sie nur für Zeiger als Typen für T, weil das delete in Destruktor für andere Datentypen einen Fehler erzeugt. Die nachfolgenden Beschreibungen sind also rein für Zeiger ausgelegt. Die Objekte auf die die Zeiger zeigen müssen außerdem am *heap* angelegt worden sein.

Der Konstruktor der Klasse *NameListMap* hat keinerlei spezielles Verhalten und erzeugt nur den Container für die Variablennamen bzw. -werte. Der Destruktor hingegen besitzt die spezielle Aufgabe die Objekte, welche sich hinter den Zeigern verstecken zu löschen. Der Container kann also nur mit Zeigern als Value verwendet werden. Der Destruktor iteriert also über den gesamten Container und löscht jeden Value jedes Keys und setzt den Zeiger auf den *nullpointer*.

Beim Abfragen einer Variable überprüft die Funktion im ersten Schritt, ob die Variable im Vorhinein registriert wurde, also ob der Variablenname als Schlüssel in der map vorhanden ist. Sollte er die nicht sein, so wird ein Fehler geworfen. Anschließend wird der Wert (in unserem Fall der entsprechende Zeiger) zurückgegeben. Sollte kein Schlüssel existieren wird automatisch ein *nullpointer* zurückgegeben.

Beim Registrieren einer Variable mit einem Wert wird überprüft, ob die Variable bereits existiert. Sollte dies der Fall sein, wird der alte Wert (der alte Zeiger) gelöscht und mit dem neuen ersetzt. Wenn der Schlüssel noch nicht vorhanden ist, wird ein neues Element zu dem Container hinzugefügt.

Soll die Liste ausgegeben werden, wird über den Container iteriert und jeder Schlüssel mit seinem zugehörigen Wert ausgegeben.

KNOTEN FÜR SYNTAXBAUM

Die Klassehierarchie mit der Basisklasse *StNode* repräsentiert die verschiedenen Arten von Knoten, aus welchen ein Syntaxbaum bestehen kann. Jede der drei Knotenarten besteht hierbei aus zwei Zeigern, welche wiederum auf zwei neue Knoten zeigen können. Für jeden Knoten besteht außerdem die Möglichkeit mithilfe von entsprechenden Methoden die Objekte hinter den Zeigern neu zu setzen.

Sollte ein Knoten gelöscht werden wollen, werden zuerst die Objekte hinter den Kinderknoten zerstört und anschließend die Zeiger zu *nullpointer* verändert.

Unterschiede in den Implementierungen der verschiedenen Knotenarten gibt es nur bei den Methoden für das Ausgeben und Auswerten eines Knoten:

- Werteknoten: Dieser Knoten besteht neben den beiden Zeigern auf seine Kinder aus einem Wert, den dieser Knoten repräsentieren soll. Wird der Knoten ausgegeben, wird nur der entsprechende Wert ausgegeben und wird der Knoten ausgewertet, wird nur der Wert retourniert, da keine Rechnung erforderlich ist.
- Operatorknoten: Dieser Knoten besteht neben den Zeigern aus einem Operator für eine Rechnung (+, -, *, / oder ^). Wird der Knoten ausgegeben wird der entsprechende Operator ausgegeben, wird der Knoten jedoch ausgewertet, wird zuerst der linke Kinderknoten ausgewertet und durch den Operator mit dem rechten Kinderknoten verbunden und das Ergebnis retourniert.
- Identifikernoten: Dieser Knoten repräsentiert eine Variable und besteht neben den Zeigern aus einer Zeichenkette, die den Variablennamen beinhaltet. Sollte der Knoten nun ausgewertet werden, wird in der Namensliste der entsprechende Wert für den Variablennamen gesucht und der Baum hinter der Variable ausgewertet. Das Ergebnis wird retourniert. Ausgegeben wird nur der Variablenname.

SYNTAXBAUM

Der Syntaxbaum selbst wird durch die Klasse *SyntaxTree* bereitgestellt. Der Baum besitzt einen Zeiger auf seinen Wurzelknoten. Wird der Baum gelöscht wird die Löschmethode für den Wurzelknoten aufgerufen, woraus eine rekursive Löschung aller Knoten, die mit dem Baum verbunden sind, resultiert. Das Evaluieren eines Baumes funktioniert auf dieselbe Weise mithilfe der Evaluierungsmethode des Wurzelknotens, welche bei Bedarf ebenfalls rekursiv alle anderen Knoten evaluiert.

Der Baum bietet außerdem die Möglichkeit den Wurzelknoten neu zu setzen. Für das Ausgeben eines Baumes wird ein weiteres Mal die entsprechende Methode des Wurzelknotens aufgerufen. Der Operator << wurde für einen Zeiger auf einen Baum überladen, um ein leichteres Ausgeben des Baumes zu ermöglichen.

CODE

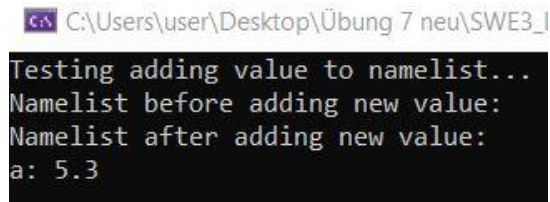
Siehe Visual Studio

TESTFÄLLE

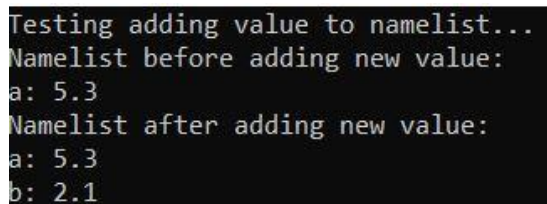
Aufgrund eines Fehlers beim Zerstören von Objekten, den ich nicht beheben konnte, bitte ich um Rückmeldung der Fehlerquelle und kann daher die Testfälle nur beschreiben, da das Programm bei Aufruf des ersten Destruktors immer abbricht.

NAMENSLISTE

Testfall 1: Hinzufügen von Variablen zu der Namensliste



```
C:\Users\user\Desktop\Übung 7 neu\SWE3_1>
Testing adding value to namelist...
Namelist before adding new value:
Namelist after adding new value:
a: 5.3
```



```
Testing adding value to namelist...
Namelist before adding new value:
a: 5.3
Namelist after adding new value:
a: 5.3
b: 2.1
```

Testfall 2: Überschreiben einer bereits existierenden Variable

Testfall 3: Ausgabe einer leeren Liste

Testfall 4: Abfragen eines Values

Testfall 5: Abfragen eines nicht vorhandenen Values

NODE

Testfall 1: Erstellen eines Werteknotens

Testfall 2: Erstellen eines Operatorknotens

Testfall 3: Erstellen eines Identifiknotens

Testfall 4: Setzen eines left und right

Testfall 5: Ausgabe eines Werteknotens

Testfall 6: Ausgabe eines Operatorknotens

Testfall 7: Ausgabe eines Identifiknotens

Testfälle 8-10: Auswertungen verschiedener Knoten

Testfall 11: Auswertung eines Knotens mit nicht registriertem Identifier

Testfall 12: Auswertung einer Division durch 0

SYNTAX TREE

Testfall 1: Erstellen eines Baumes

Testfall 2: Setzen eines Wurzelknotens

Testfall 3: Auswerten eines Baumes mit Ausdruck der Division durch 0 enthält

Testfall 4: Ausgabe eines Baumes

Testfall 5: Ausgabe eines leeren Baumes

Testfälle 6-... : Auswerten verschiedener Ausdrücke

PARSER

Testfall 1: Testen eines Set Befehles

Testfall 2: Testen eines Print Befehles

Testfall 3: Testen mehrerer Befehle hintereinander

Testfall 4 und 5: Set und Print mit Division durch 0

Testfall 6: Parsen eines leeren Files

Testfall 7: fehlerhafter Term und Faktor

ZEITAUFWAND

20+ Stunden