252-0027-00: Einführung in die Programmierung

Übungsblatt 12

Abgabe: 17. Dezember 2019, 10:00

Checken Sie die neue Übungs-Vorlage aus. Vergessen Sie nicht, Tests zu schreiben!

Aufgabe 1: Interpreter

In der letzten Übung implementierten Sie einen Evaluator für mathematische Ausdrücke. In dieser Aufgabe erweitern Sie ihn so, dass er statt einzelnen Ausdrücken einfache Programme mit mehreren Anweisungen auswertet. Das nennt man auch *interpretieren* und ein solches Programm entsprechend *Interpreter*.

```
digit  ⇐ 0 | 1 | ... | 9
char   ⇐ A | B | ... | Z | a | b | ... | z
num    ⇐ digit { digit } [ . digit { digit } ]
var    ⇐ char { char }
func   ⇐ char { char } (
op     ⇐ + | - | * | / | ^
atom   ⇐ num | var
term   ⇐ ( expr ) | func expr ) | atom
expr   ⇐ term [ op term ]
stmt   ⇐ var = expr ;
prog   ⇐ { stmt }
```

Abbildung 1: **EBNF-Beschreibung** von *prog*

```
alpha = i * ((2*PI) * (1 / 6.05));
size = (0.25 * cos(t/2)) + 0.75;

x = cos(alpha + (0.3 * t)) * size;
y = sin((1.5 * alpha) + t) * size;

r = (cos(alpha + (2 * t)) + 1) / 2;
g = (sin(alpha + (2 * t)) + 1) / 2;
b = (cos(alpha +  (PI/2)) + 1) / 2;
```

Abbildung 2: Beispiel-Programm

Abbildung 1 zeigt die EBNF-Beschreibung für Programme (*prog*). Es gibt nur eine Art von Anweisung (*stmt*), nämlich eine Zuweisung eines Ausdrucks zu einer Variable. Beachten Sie, dass die Beschreibung zugunsten der Lesbarkeit nicht alle Tokenizer-Regeln explizit enthält: die RHS der Parser-Regeln enthalten teilweise auch direkt Buchstaben (`blau`). Der aktualisierte `Tokenizer` in der Übungsvorlage stellt aber auch für die Buchstaben `( ) = ;` die benötigten Methoden zur Verfügung.

In der Vorlage befindet sich die `Interpreter`-Klasse, welche (abgesehen von Klassen- und Methodennamen) dem fertigen `ExprEvaluator` der letzten Übung entspricht. Die `Interpreter`-

Klasse befindet sich in einem *Paket* (engl. package) namens `language`. Platzieren Sie alle Klassen, welche Sie für diese Aufgabe erstellen, ebenfalls in diesem Paket.

a) Ändern Sie den `Interpreter` so, dass er Programme, die der Beschreibung in Abbildung 1 entsprechen, interpretiert. Die Semantik von Zuweisungen soll dieselbe sein wie in Java.

   Aufgrund der Möglichkeit, Variablen zu definieren, reicht die einfache rekursive Struktur des `ExprEvaluators` nicht aus, um Programme zu interpretieren. Der Interpreter muss sich zusätzlich den Zustand des Programms, d.h. die definierten Variablen und deren Werte, merken. Dafür eignet sich eine `Map`, oder genauer, da wir eine Abbildung von Variablen-Namen auf Werte brauchen, eine `Map<String,Double>`. Erstellen Sie eine solche im `Interpreter`-Konstruktor und verwenden Sie sie wo nötig. Neu könnte der Konstruktor auch eine `Map` von Variablennamen und -werten entgegennehmen anstatt dem einzelnen Wert für die x-Variable.

   Schreiben Sie die fehlenden `interpret*(...)`-Methoden und ändern Sie `interpret(String)` entsprechend. Beachten Sie, dass Anweisungen (*stmt*s) im Gegensatz zu Ausdrücken (*expr*s) selbst keinen Wert haben, sondern nur einen Effekt auf den Programmzustand (z.B. die Änderung eines Wertes einer Variable). Ein Programm als Ganzes hat ebenfalls keinen Wert, also können Sie den Rückgabetyp von `interpret(String)` auf `void` ändern.

b) Schreiben Sie ein Java-Programm `Repl` (im Paket `language`), welches eine *REPL* implementiert. "REPL" steht für *read–eval–print loop* und bezeichnet ein (Java-)Programm, welches wiederholt Anweisungen von der Konsole liest (*read*), diese auswertet (*eval*) und das Resultat ausgibt (*print*). Der Programmzustand (d.h. die Werte von Variablen) wird über mehrere Anweisungen hinweg mitgeführt und durch das Interpretieren von Anweisungen verändert.

   Sie können sich an der `EvaluatorApp` der letzten Übung orientieren. Da Programme selber keinen Wert haben, gibt es kein explizites Resultat zum Ausgeben. Geben Sie stattdessen nach jeder Ausführung alle definierten Variablen und deren Werte aus. Dafür brauchen Sie Zugriff auf die Variablen-`Map` des Interpreters. Achten Sie darauf, dass die `ProgramException` richtig behandelt wird.

Aufgabe 2: Programmatisches Zeichnen

Mit Ihrem Interpreter können Sie noch interessantere Dinge zeichnen als die einfachen $f(x)$-Funktionen der letzten Übung. In der Vorlage finden Sie eine `DrawingApp`, welche ein GUI mit einem Eingabefeld für ein Programm und einem für eine Anzahl Repetitionen erstellt. Die Idee ist, dass das Programm wiederholt ausgeführt wird und dass dieses in jeder Iteration ein (x, y)-Koordinaten-Paar und zusätzlich ein (r, g, b)-Farbtripel liefert (r, g und b sind dabei jeweils zwischen 0.0 und 1.0). Nach jeder Repetition[1] wird eine Linie zwischen dem alten und dem neuen Koordinaten-Paar mit der neuen Farbe gezeichnet. Als Input, d.h. als Start-Programmzustand, bekommt das Programm Werte für folgende Variablen:

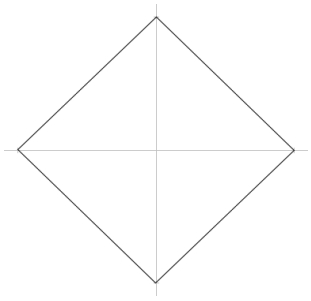
`n`	die Anzahl Repetitionen
`i`	die aktuelle Iteration ($0 \leq$ `i` $\leq$ `n`)
`x`, `y`, `r`, `g`, `b`	die Resultate der letzten Ausführung (in der ersten Iteration: 0.0)
`PI`, `E`, ...	mathematische Konstanten

[1] D.h. nach jeder Iteration ausser der ersten – die Anzahl Iterationen ist also um 1 grösser als die Anzahl Repetitionen!

Mit diesem Mechanismus lassen sich verschiedenste Formen zeichnen. Ein Programm, das ein gleichmässiges n-Eck zeichnet, sieht z.B. so aus:

```
alpha = i * ((2*PI) / n);
x = cos(alpha);
y = sin(alpha);
```

Für $n = 4$ liefert dieses Programm die Koordinaten-Paare $(1, 0)$, $(0, 1)$, $(-1, 0)$, $(0, -1)$, welche zu folgendem Quadrat verbunden werden:



Im Anhang finden Sie weitere Programme und die daraus resultierenden Zeichnungen.

a) Vervollständigen Sie die `run()`-Methode in `DrawingApp`. Es ist bereits Code vorhanden, der Arrays für die x- und y-Koordinaten und deren Farben erstellt. Die `drawLines()`-Methode nimmt diese Arrays entgegen und zeichnet sie. Was noch fehlt, ist der Code, der diese Werte berechnet. Erstellen Sie einen `Interpreter`, initialisieren Sie die oben aufgeführten Variablen, führen Sie das im Feld `program` gespeicherte Programm wiederholt aus und lesen Sie nach jeder Iteration die Variablen-Werte aus. Jede Ausführung übernimmt dabei den Programmzustand der vorhergehenden Ausführung. Einzige Ausnahme: Die Variable i müssen Sie vor jeder Ausführung aktualisieren!

b) Die Schleife der Methode `run()` zeichnet das Bild immer wieder neu. Das erlaubt uns, animierte Zeichnungen zu machen! Dazu ist nur eine zusätzliche Variable t nötig, welche vor jeder Ausführung von `program` mit der aktuellen Zeit initialisiert wird[2].

   Wir definieren die "aktuelle Zeit" als die Anzahl Sekunden, die seit der letzten Änderung des Programms verstrichen sind. Diese erhalten Sie, indem Sie in `setProgram()` mit `System.currentTimeMillis()` den Zeitpunkt "0" speichern und in `run()` die Differenz zur aktuellen Zeit berechnen. Beachten Sie, dass t `double`-Genauigkeit haben soll.

   Um eine Zeichnung zu animieren, verwenden Sie t an einem geeigneten Ort, z.B. so:

   ```
   alpha = i * ((2*PI) / n);
   x = cos(alpha + t);
   y = sin(alpha + t);
   ```

   Geben Sie auch mal das Beispiel-Programm in Abbildung 2 ein und setzen Sie die Anzahl der Repetitionen auf 121, um zu sehen, welch interessante Animationen möglich sind.

[2] Sie können den selben Wert für t für alle Iterationen verwenden.

Aufgabe 3: Compiler

xkcd: Compiling by Randall Munroe (CC BY-NC 2.5)

Wie Sie beim Herumspielen mit der `DrawingApp` vielleicht festgestellt haben, kommt das Programm bei vielen Repetitionen an seine Grenzen. Im Panel unten rechts werden die *FPS* (frames per second) angezeigt. Dies sind die Anzahl Bilder, die das Programm pro Sekunde zeichnen kann. Wenn dieser Wert zu klein wird, sieht die Animation nicht mehr flüssig aus.

Das Problem ist, dass das Interpretieren von Quellcode ineffizient und langsam ist. Deshalb werden Java-Programme auch zuerst *kompiliert*, bevor sie ausgeführt werden. Kompilieren heisst, den Quellcode in eine Form zu übersetzen, die vom Computer direkt(er) ausgeführt werden kann. In dieser Übung schreiben Sie einen einfachen Compiler, der den Quellcode von Programmen von Aufgabe 1 in eine Serie von Instruktionen übersetzt, die effizient ausgeführt werden können.

Die Programmiersprache in Aufgabe 1 hat eine rekursive Struktur: Ausdrücke können mehrere Ausdrücke enthalten, welche wiederum mehrere Ausdrücke enthalten können, usw. Um eine solche Struktur in eine lineare Folge von Instruktionen umzuwandeln, verwenden wir einen *Operanden-Stack*. Dies ist ein `Stack` (wie Sie Ihn in der Vorlesung gesehen haben), der Zwischenresultate von Berechnungen speichert. Instruktionen können Werte auf den Stack "pushen" oder Werte ab dem Stack "poppen" und verwenden. Es gibt folgende Arten von Instruktionen:

`CONST` c	Pusht den konstanten Wert c auf den Stack
`LOAD` v	Lädt den Wert der Variable v und pusht ihn auf den Stack
`STORE` v	Poppt einen Wert vom Stack und speichert ihn in der Variable v
`OP` $\oplus$	Poppt zwei Werte r und l vom Stack (zuerst r, dann l) berechnet $l \oplus r$ und pusht das Resultat zurück auf den Stack
`FUNC` f	Poppt einen Wert x vom Stack, berechnet $f(x)$ und pusht das Resultat zurück auf den Stack

Unten sehen Sie ein kleines Programm, das aus solchen Instruktionen besteht. Es lädt zuerst den Wert der Variable `x` und dann einen konstanten Wert 2 auf den Stack. Die nächste Instruktion holt sich die beiden Werte vom Stack, multipliziert sie und pusht das Resultat zurück. Die letzte Instruktion schliesslich holt diesen Wert vom Stack und speichert ihn zurück in die Variable `x`:

```
LOAD x
CONST 2
OP *
STORE x
```

Sie sollen nun einen `Compiler` schreiben, welcher ein Programm in eine Liste solcher Instruktionen kompiliert. Der `Compiler` geht grundsätzlich gleich vor wie der `Interpreter`: er parst das Programm rekursiv und berechnet gleichzeitig ein Resultat. Im Gegensatz zum `Interpreter` berechnet er aber keine Werte, sondern generiert Listen von Instruktionen.

Um zu verstehen, wie diese Instruktionen genau generiert werden, betrachten Sie Tabelle 1,

Programmteil	Instruktionen
`b`	`LOAD b`
`1`	`CONST 1`
`b + 1`	`LOAD b CONST 1 OP +`
`(b + 1)`	`LOAD b CONST 1 OP +`
`2`	`CONST 2`
`c`	`LOAD c`
`2 * c`	`CONST 2 LOAD c OP *`
`sin(2 * c)`	`CONST 2 LOAD c OP * FUNC sin`
`(b + 1) / sin(2 * c)`	`LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP /`
`a = (b + 1) / sin(2 * c);`	`LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP / STORE a`

Tabelle 1: Kompilieren der Anweisung `a = (b + 1) / sin(2 * c);`

welche zeigt, wie der Ausdrucks `a = (b + 1) / sin(2 * c);` kompiliert wird. In der linken Spalte stehen die Programmteile (in der Reihenfolge, in der sie auch schon vom Interpreter fertig geparst werden) und in der rechten Spalte die entsprechenden Instruktionen. Zum Beispiel sehen Sie, dass der `Compiler` für die Zahl `1` im Quellcode die Instruktion `CONST 1` generiert. Oder, dass er für einen binären Ausdruck zuerst die Instruktionen des linken Teils, dann die Instruktionen des rechten Teils und schliesslich eine `OP`-Instruktion generiert.

a) Erstellen Sie zuerst einige Klassen `ConstInstr`, `LoadInstr`, usw. für die verschiedenen Arten von Instruktionen. Erstellen Sie dazu am besten ein Unterpaket `language.instructions` und deklarieren Sie die `*Instr`-Klassen als `public`. Für die `OP`- und `FUNC`-Instruktionen können Sie entscheiden, ob Sie jeweils eine einzige Klasse verwenden möchten oder mehrere Subklassen, eine für jeden unterstützten Operator, bzw. für jede Funktion.

   Alle Instruktionen sollen ein Interface `Instr` (ebenfalls im `language.instructions`-Paket zu erstellen) mit einer `execute()`-Methode implementieren. Diese Methode nimmt als Argumente den Operanden-`Stack` und die Variablen-`Map` und führt die Instruktion aus. Falls ein Fehler auftritt (z.B., wenn eine Variable nicht definiert ist oder wenn der Stack nicht die erwartete Grösse hat), soll die Methode eine Ausnahme der Klasse `ExecutionException` (die Sie erstellen) werfen. Diese Art von Exception soll *checked* sein.

b) Erstellen Sie eine `Program`-Klasse, welche eine Liste von Instruktionen enthält und eine `execute()`-Methode hat, welche diese Instruktionen ausführt. Diese Methode nimmt eine Variablen-`Map` entgegen, die vom Programm verwendet und aktualisiert wird.

   Das Ausführen der Instruktionen ist denkbar einfach: Erst wird ein Operanden-Stack erstellt, und dann wird eine Instruktion nach der anderen über ihre `execute()`-Methode ausgeführt.

c) Schreiben Sie jetzt die `Compiler`-Klasse. Sie können sie analog zur `Interpreter`-Klasse entwerfen, mit `compile*()`- statt `interpret*()`-Methoden. Jede dieser Methoden soll eine Liste zurückgeben, welche die Instruktionen enthält, die dem geparsten Programm-Teil entsprechen. Die Haupt-Methode `compile(String)`, schliesslich, soll alle Instruktionen in eine `Program`-Instanz packen und diese zurückgeben.

d) Ändern Sie zum Schluss die `DrawingApp` so, dass sie statt dem `Interpreter` den `Compiler` verwendet, um die eingegebenen Programme auszuführen. Sinnvollerweise sollten Sie das Programm in `setProgram()` kompilieren, und zwar nur wenn es sich verändert hat. In `run()`

wird das Programm dann "nur" noch ausgeführt. Sie sollten aber weiterhin Fehler beim Kompilieren und beim Ausführen abfangen und die Fehlermeldung auf dem Panel ausgeben.

Wenn Sie alles richtig gemacht haben, sollten Sie jetzt Zeichnungen mit einer deutlich höheren Anzahl an Repetitionen flüssig animieren können!

Aufgabe 4: Wörter (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin.

In dieser Aufgabe implementieren Sie verschiedene Analysen von Wörtern eines Textes. Ein Wort eines Textes wird von der Klasse `MyWord` repräsentiert. Diese Klasse speichert sowohl das Wort als String, als auch alle Positionen des Wortes im Text. Zum Beispiel in dem Text "Seit Wochen wusste er, dass er die Prüfung nicht bestehen würde." hat das Wort "Seit" die einzelne Position 0 und "er" die Positionen 3 und 5. Zwei Wörter sind unterschiedlich, wenn die Wörter als unterschiedliche Strings im Text vorkommen. So sind "verlieren" und "verliert" unterschiedliche Wörter. Des Weiteren definieren wir eine Ordnung auf Wörtern. Ein Wort `x` ist kleiner als ein Wort `y`, wenn der Abstand zwischen der ersten und letzten Position von `x` im Text kleiner ist als der Abstand zwischen der ersten und letzten Position von `y` im Text.

Ihre Aufgabe ist es nun, die Klasse `WordService` und ihre Analysen zu implementieren. Die Klasse hat einen Konstruktor, welche einen Text aus einem `WordScanner` ausliest und damit das Objekt initialisieren soll. Ein `WordScanner` funktioniert ähnlich wie `Scanner`, mit dem Unterschied, dass `next()` Satzzeichen aus dem nächsten String entfernt bevor es den String zurückgibt. Sie müssen die Wörter nicht weiter bearbeiten und dürfen annehmen, dass jeder von `next()` zurückgegebener String ein eigenes Wort ist.

a) Implementieren Sie die Methode `getWords`, welche alle Wörter des Texts als `MyWord` Objekte zurückgibt.

b) Implementieren Sie nun die Methode `top`, welche einen Integer `number` als Parameter nimmt und die `number` (Anzahl) grössten Wörter zurückgibt. Beachten Sie, dass Grösse durch unsere zuvor auf Wörtern definierte Ordnung bestimmt ist. Die Ergebnisliste muss absteigend sortiert sein, wobei die Reihenfolge von gleich grossen Worten egal ist.

Achtung: Ihre implementierten Methoden dürfen nur `IllegalArgumentException` werfen. Das Werfen anderer Exceptions ist verboten, unabhängig von den verwendeten Argumenten. Tests sind in der Datei "WordServiceTest.java" enthalten.

Aufgabe 5: Hoare Triple

Welche dieser Hoare Tripel sind (un)gültig? Bitte geben Sie für ungültige Tripel ein Gegenbeispiel an. Die Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow.

1. `{ x >= 0 || y >= 0 }  z = x * y;  { z > 0 }`

2. `{ x <= 0 && y >= 0 && x = y }  z = x * y;  { z <= 0 }`

3. `{x > 10 }  z = x % 10;  { z > 0 }`

4. `{x > 0 }  y = x * x; z = y / 2;  { z > 0 }`

5. `{ true }`

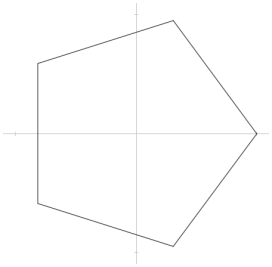
```
if (x > y) {
    y = x;
} else {
   y = - x;
}

{ y >= x }
```

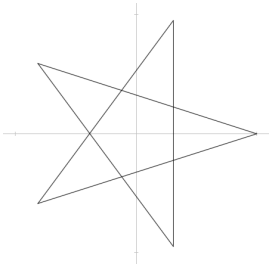
6. `{ b > c  }`

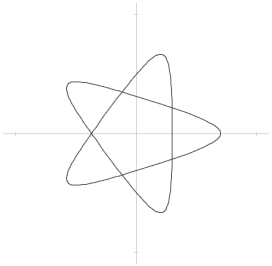
```
if (x > b) {
    a = x;
} else {
    a = b;
}

{ a > c }
```

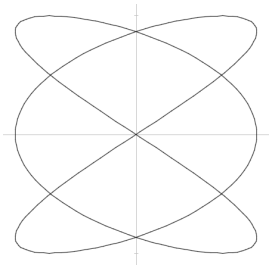
Anhang: Programme und Zeichnungen

Bezeichnung	Programm	n	Zeichnung

n-Eck	`alpha = i * ((2*PI) / n);`
	`x = cos(alpha);`
	`y = sin(alpha);`
		$n \geq 3$	

n-Stern	`m = 2;`
	`alpha = i * ((2*PI) * (m/n));`
	`x = cos(alpha);`
	`y = sin(alpha);`
		$n \geq 5$,
		n ungerade
			

Hypotrochoid
(Wikipedia)
	`rA = 0.5;`
	`rB = 0.3;`
	`d = 0.5;`
	`theta = i * ((2*PI) / (n/3));`
	`diff = rA - rB;`
	`x = (diff * cos(theta))`
	`  + (d * cos((diff / rB) * theta));`
	`y = (diff * sin(theta))`
	`  - (d * sin((diff / rB) * theta));`
		grosse n	

Lissajous-Figur
(Wikipedia)
	`pA = 3;`
	`pB = 2;`
	`alpha = i * ((2*PI) / n);`
	`x = cos(pA * alpha);`
	`y = sin(pB * alpha);`
		grosse n	

Kombination
(mit `signum`-Trick)
	`alphaA = (i * ((2*PI) / 200)) + (PI/2);`
	`xA = cos(alphaA);`
	`yA = sin(alphaA);`
	`alphaB = (i * ((2*PI) / 3)) + ((7/6)*PI);`
	`xB = cos(alphaB);`
	`yB = sin(alphaB);`
	`caseB = ((signum(i-200.5) + 1) / 2);`
	`caseA = 1 - caseB;`
	`x = (caseA * xA) + (caseB * xB);`
	`y = (caseA * yA) + (caseB * yB);`
		$n = 203$