Programmieren in Java

Vorlesung 10: Ein Interpreter für While

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2015

Inhalt

Interpreter für While

Interpreter

- Ein Interpreter ist ein Programm zur Ausführung von anderen Programmen
- Eingaben
  - Ein Programm (-text) P in einer Sprache L
  - Die Eingabe für das Programm P
- Ausgabe
  - Die Ausgabe des Programms P

Architektur des Interpreters

- Parser
  - Einlesen des Programmtextes
  - Erstellen einer Objektrepräsentation des Programmtextes *abstract syntax tree* (AST)
- Runner
  - Durchlaufen des AST
  - Ausführen der Statements
  - Auswerten von Ausdrücken

Die Sprache While

Syntax von While

BNF für While

$\langle stmt \rangle$	::=	$\langle var \rangle = \langle expr \rangle$	Zuweisung
	|	`if (` $\langle expr \rangle$ `)` $\langle stmt \rangle$ `else` $\langle stmt \rangle$	Bedingte Anweisung
	|	`while (` $\langle expr \rangle$ `)` $\langle stmt \rangle$	While Schleife
	|	`{` $\langle stmt1 \rangle$	Sequenz
$\langle stmt1 \rangle$	::=	`}`	. . . leer
	|	$\langle stmt \rangle$ `;` $\langle stmt1 \rangle$	nächste Anweisung

- $\langle stmt \rangle$ = Statement = Anweisung
- $\langle stmt1 \rangle$ spezifiziert eine Liste von Anweisungen, wobei jede Anweisung durch Semikolon abgeschlossen ist und das Ende mit schließender geschweifter Klammer markiert ist
- $\langle expr \rangle$ = arithmetische Ausdrücke, wie gehabt

Objektrepräsentation (AST)

- Composite und Visitor Pattern
- Hier: Visitor *ohne* Rückgabewert

```
1 public interface IStmt {
2   void accept(StmtVisitor v);
3 }
```

```
1 public interface StmtVisitor {
2   void visitAssign(Assign s);
3   void visitIf(If s);
4   void visitSequence(Sequence s);
5   void visitWhile(While s);
6 }
```

Objektrepräsentation (AST) II

```
1 public class Assign implements IStmt {
2   public final String var;
3   public final IExpr exp;
4   public void accept(StmtVisitor v) {
5     v.visitAssign(this);
6   }
7 }
```

```
1 public class While implements IStmt {
2   public final IExpr condition;
3   public final IStmt body;
4   public void accept(StmtVisitor v) {
5     v.visitWhile(this);
6   }
7 }
```

```
1 public class If implements IStmt {
2   public final IExpr condition;
3   public final IStmt trueBranch;
4   public final IStmt falseBranch;
5   public void accept(StmtVisitor v) {
6    v.visitIf(this);
7   }
8 }
```

```
1 public class Sequence implements IStmt {
2   public final Collection<IStmt> stmts;
3   public void accept(StmtVisitor v) {
4     v.visitSequence(this);
5   }
6 }
```

- Für jede Art Anweisung eine Klasse (mit Standardkonstruktor)

Parser für While

Parser für While

- Gleiches Muster wie für Ausdrücke:
  recursive descent parser
- Möglichkeit: Erweiterung des Ausdrucksparsers mittels Vererbung
- Kleine Änderungen am Ausdrucksparser lecture20150622.Parser erforderlich:
  die privaten Methoden für den Scanner müssen protected gemacht werden

```
1 protected boolean lookingAt(String regex) {
2   return scan.lookingAt(regex);
3 }
4 protected String getLexeme(String regex) {
5   return scan.getLexeme(regex);
6 }
```

Parser für If-Anweisung (Beispiel, Auszug)

```
1  public IStmt parseStmt() {
2    String lexeme = scan.getLexeme(REGEX_WORD);
3    if ("if".equals(lexeme)) {
4      // "if" . "(" <expr> ")" <stmt> "else" <stmt>
5      if (scan.getLexeme(REGEX_OPEN_PAREN) != null) {
6        // "if" "(" . <expr> ")" <stmt> "else" <stmt>
7        IExpr condition = parseExpr();
8        // "if" "(" <expr> . ")" <stmt> "else" <stmt>
9        if (scan.getLexeme(REGEX_CLOSE_PAREN) != null) {
10         // "if" "(" <expr> ")" . <stmt> "else" <stmt>
11         IStmt trueBranch = parseStmt();
12         // "if" "(" <expr> ")" <stmt> . "else" <stmt>
13         if (scan.getLexeme("else") != null) {
14           // "if" "(" <expr> ")" <stmt> "else" . <stmt>
15           IStmt falseBranch = parseStmt();
16           // "if" "(" <expr> ")" <stmt> "else" <stmt> .
17           return new If(condition, trueBranch, falseBranch);
18 } } } }
```

Ausführung von While

Ausführung von While

- Durch RunVisitor, eine Implementierung von StmtVisitor
- Interner Zustand des RunVisitor ist die Belegung der Variablen
- Repräsentiert durch eine Abbildung von Variablennamen auf Zahlen: Map<String,Integer>

RunVisitor

```
1  public class RunVisitor
2      implements StmtVisitor {
3
4    public final Map<String,Integer> state;
5    private final IExprVisitor eval;
6
7    public RunVisitor(
8        Map<String,Integer> state) {
9      this.state = state;
10     this.eval = new Eval(state);
11   }
12
13   public void visitAssign(Assign s) {
14     int value = s.exp.accept(eval);
15     state.put(s.var, value);
16   }
```

```
1    public void visitIf(If s) {
2      if (s.condition.accept(eval) != 0) {
3        s.trueBranch.accept(this);
4      } else {
5        s.falseBranch.accept(this);
6      }
7    }
8    public void visitSequence(Sequence s) {
9      for(IStmt stmt : s.stmts) {
10       stmt.accept(this);
11     }
12   }
13   public void visitWhile(While s) {
14     while( s.condition.accept(eval) != 0) {
15       s.body.accept(this);
16     }
17   }
18 }
```

Testen des Interpreters

Testen des Interpreters

- Mittels JUnit Testfällen
- Für jeden Testfall muss zunächst eine Belegung der Variablen, also eine Map, generiert werden und damit ein RunVisitor instanziert werden:

```
1 Map<String,Integer> state = new HashMap<String,Integer>();
2 RunVisitor r = new RunVisitor(state);
```

- Anstatt diese Anweisungen in jedem Testfall zu wiederholen, kann eine *Before-Methode* geschrieben werden.
- Dafür müssen die Werte in Instanzvariable der Testklasse verlagert werden.

Before-Methode

```
1  public class RunVisitorTest {
2    private Map<String, Integer> state;
3    private RunVisitor r;
4    @Before public void setup() {
5      state = new HashMap<String,Integer>();
6      r = new RunVisitor(state);
7    }
8    @Test public void test() {
9      IStmt ass1 = new Assign("x", cnst(1));
10     ass1.accept(r);
11     assertEquals(state.get("x"), new Integer(1));
12   }
13 }
```

Vor **jedem** Testfall mit Annotation @Test

- wird eine neue RunVisitorTest Instanz erzeugt
- werden die Methoden mit Annotation @Before ausgeführt