Eine kleine Skriptsprache

Sven Eric Panitz

13. Januar 2022

Inhaltsverzeichnis

1. Neuere Konstrukte Java

In dieser Aufgabe werden eine Vielzahl von Java-Features verwendet, die erst später teilweise erst 2021 Einzug in die Sprache gefunden haben, und die Sprache um Features, die nicht allein der Objektorientierung zuzuordenen sind, erweitert.

Die Konstrukte, die wir verwenden sind im einzelnen:

- Aufzählungsklasse, wie sie seit Java 5 in der Sprache sind. Damit sind sie zwar kein neues Feature, aber sind eine der Erweiterungen, die nicht als objektorientiertes zu betrachten sind. (Schlüsselwort `enum`.)
- Lambda-Ausdrücke, die mit der version 8 in die Sprache Java eingezogen sind. (Symbol: `->`.)
- Standardmethoden in Schnittstellen. (Schlüsselwort `default`.)
- Recordklassen, mit denen Java ab Version 16 erweitert wurde. (Schlüsselwort `record`.)
- Switch-Ausdrücke, die seit Java 16 verfügbar sind. Inklusive des Schlüsselwortes `yield`.
- Pattern-Match in Switch, das seit Java 17 ein Preview Feature ist.
- Versiegelte Schnittstellen/Klassen. (Schlüsselwörter `sealed` und `permits`.)

2. Skriptsprache

All diese Eigenschaften werden wir auf engsten Raum verwenden, um eine kleine Skriptsprache zu implementieren. Damit setzen wir eine erste kleine Aufgabe im Bereich Compilerbau um.

Wir wollen eine kleine Skriptsprache realisieren, die auf Daten des datentyps `long` rechnet.

Die Sprache soll folgende Konstrukte kennen:

- Zahlenkonstanten: `42`
- Variablen: `x`
- Zuweisung an Variablen: `x = 42`
- arithmetische Operatorausdrücke: `42 * x`
- Fallunterscheidungen: `if y then 42 else 0`
- eine Schleife: `while y do y = y - 1`
- Eine Sequenz von Anweisungen: `{x = 42  y = 17  z = 4}`

Wir verzichten für die Aufgabe darauf, dass auch Funktionen definiert und aufgerufen werden können.

Ein kleines Beispielskript zur Berechnung der Falkultät von 5, sieht in der Sprache wie folgt aus:

```
{
  x = 5
  r = 1
  while x do {
     r = r*x
     x = x-1
  }
  r
}
```

Da wir als Datentyp nur den Typ `long` verwenden, so wird in der Bedingung der Schleife, die Zahl 0 als `false` interpretiert und alle anderen Werte als `true`.

Wir geben in dieser Aufgabe allerdings keine konkrete Oberflächensyntax der Sprache, sondern nur das interne Datenmodell.

Für einen kompletten Interpreter der Skriptsprache fehlt dann noch ein Parser, der aus einem Programmtext die entsprechende Datenstruktur erzeugt. Dieser ist im Anhang dieses Papiers mit dem Compiler Generator Tool ANTLR definiert.

3. Schnittstelle für eine kleine Skriptsprache

Es gibt ein paar Standardklassen, die wir für unsere Implementierung verwenden werden:

Java: Expr

```java
package name.panitz.util;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;
import java.util.List;
import java.util.function.BinaryOperator;
import java.io.Reader;
import java.io.StringReader;
```

Zwei weitere Klassen der Laufzeitumgebung des generierten Parsers unserer Skriptsprache aus dem Anhang dieses Papiers werden benötigt. Hierzu ist die enztsprechende Jar-Datei in dem Klassenpfad mit aufzunehmen.

Java: Expr

```
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.CommonTokenStream;
```

Die Klassen gehören zu dem Compilerbau-Tool ANTLR. Sie müssen sich hierzu folgende JAR-Datei laden und zu ihren Klassenpfad hinzufügen.

`https://www.antlr.org/download/antlr-4.9.3-complete.jar`

In einer Entwicklungsumgebung ist diese Datei zumeist in einem Ordner `lib` zu platzieren, auf der Kommandozeile haben der Javacompiler und der Javainterpreter die Option `-cp`, mit der der Klassenpfad gesetzt werden kann.

3.1. Die Schnittstelle für Ausdrücke der Sprache

Alle Programme der Skriptsprache werden als ein Objekt der Schnittstelle `Expr` dargestellt:

Java: Expr

```
public sealed interface Expr
```

Diese Schnittstelle soll versiegelt sein, d.h. nicht beliebige Klassen dürfen die Schnittstelle implementieren, sondern nur eine feste Anzahl von vorgegebenen Klassen.

3.1.1. Erlaubte Implementierungen

Die Klassen, die die Schnittstelle implementieren werden in einer mit `permits` eingeleiteten Auflistung angegeben. Für jedes der sieben Sprachfeatures gibt es eine Klasse. Diese Klassen sind als innere Klassen in der Schnittstelle implementiert:

Java: Expr

```
permits Expr.Var, Expr.Num, Expr.OpEx, Expr.IfEx, Expr.AssignEx,
        Expr.Seq, Expr.WhileEx {
```

Es darf also keine andere Klasse geben, die diese Schnittstelle implementiert, als die hier aufgelisteten.

3.2. Aufzählung der Operatoren

Bevor wir die innere Klassen für die einzelnen Sprachfeatures definieren, sei eine Aufzählungsklasse aller in der Skriptsprache erlaubten Operatoren definiert:

Java: Expr

```
enum Op{
  add((x,y)->x+y),
  sub((x,y)->x-y),
  mult((x,y)->x*y),
  div((x,y)->x/y),
  mod((x,y)->x%y);
```

Von dieser Aufzählungsklasse gibt es genau 5 Werte. Dieses sind die Werte `add`, `sub`, `mult`, `div`, `mod`.

In dieser Auflistung der Aufzählungswerte wird jeweils als Parameter ein Lambdaausdruck übergeben. Dieser repräsentiert die Operation, die mit dem jeweiligen Operator verbunden sein soll. Hierzu gibt es in der Aufzählungsklasse ein Feld des Typs `BinaryOperator`, in dem diese Operation gespeichert ist und es gibt einen entsprechenden Konstruktor.

Java: Expr

```
  BinaryOperator<Long> f;
  Op(BinaryOperator<Long> f){this.f = f;}
}
```

3.3. Recordklassen für die Sprachkonstrukte

Es folgt nun für jedes Sprachkonstrukt der Skriptsprache eine Recordklasse, um dieses zu repräsentieren.

In den Recordklassen wird direkt nach dem Klassennamen die Parameterliste des Konstruktors notiert. Die Parameter werden dann in finalen Feldern gleichen Namens gespeichert.

Java: Expr

```
record Var(String name) implements Expr{}
record Num(long n) implements Expr{}
record OpEx(Expr left,Op op,Expr right) implements Expr{}
record IfEx(Expr cond,Expr alt1,Expr alt2) implements Expr{}
record WhileEx(Expr cond,Expr body) implements Expr{}
record AssignEx(String v, Expr e) implements Expr{}
record Seq(List<Expr> es) implements Expr{}
```

4. Methoden mit switch-case Ausdrücken

Da ein Objekt der Schnittstelle `Expr` genau von einer der sieben angegebenen Recordklassen ist, kann man sinnvoll einen `switch-case`-Ausdruck über den Typ eines `Expr`-Objekts machen.

Seit Java 17 bietet Java dabei an, dass ähnlich wie bei einem `instanceof`-Pattern im Falle, dass ein Objekt eine Instanz von der gewünschten Klasse ist, gleich eine neue Variable eingeführt wird, die nun das Objekt als von dieser Instanz repräsentiert.

4.1. Beispiel: Aufsammeln aller Variablen

Wir machen ein erstes Beispiel. Hierzu schreiben wir eine `default`-Methode in der Schnittstelle. Sie soll in einer Menge alle Variablennamen aufsammeln, die in einem Programm der kleinen Skriptsprache auftauchen.

Java: Expr

```
default void collectVars(Set<String> vs){
```

Wir betrachten das `this`-Objekt und machen eine Fallunterscheidung mittels eines `switch-case`-Ausdrucks darüber, von welcher der sieben Recordklassen das `this`-Objekt ist.

Java: Expr

```
switch(this){
```

Der einfachste Fall ist, wenn es sich bei dem `this`-Objekt um eine einfache Variable handelt. Dann haben wir eine Variable gefunden und fügen deren Namen in die Ergebnusmenge ein.

Java: Expr

```
case Var ve -> {vs.add(ve.name);}
```

Wenn das `this`-Objekt eine Zuweisung ist, dann steht auf der linken Seite der Zuweisung eine Variable, die wir auch in Ergebnismenge einfügen. Auf der rechten Seite steht ein beliebiges weiteres Teilprogramm, für das wir auch alle dort auftretenden Variablen aufsammeln. Hierzu rufen wir die Methode rekursiv auf für die rechte Weiste der Zuweisung.

Java: Expr

```
case AssignEx ae -> {
  vs.add(ae.v);
  ae.e.collectVars(vs);
}
```

Bei einem Operatorausdruck gibt es ein Teilprogramm auf der linken und eines auf der rechten Seite des Operators. Für beide wird die Methode wieder rekursiv aufgerufen.

Java: Expr

```
case OpEx oe -> {
  oe.left.collectVars(vs);
  oe.right.collectVars(vs);
}
```

Bei einer if-Alternative gibt es gar drei Teilprogramme, für die die Methode rekursiv aufzurufen ist: die Bedingung der Fall, wenn die Bedingung nicht 0 ist und der Fall, wenn die Bedingung 0 ist. Wir erhalten drei rekursive Aufrufe.

Java: Expr

```
case IfEx ie -> {
  ie.cond.collectVars(vs);
  ie.alt1.collectVars(vs);
  ie.alt2.collectVars(vs);
}
```

Ähnlich geht es bei der Schleife. Hier gibt es neben der Bedingung aber nur einen Schleifenrumpf und keine zwei Alternativen.

Java: Expr

```
case WhileEx we -> {
  we.cond.collectVars(vs);
  we.body.collectVars(vs);
}
```

Als sechsten Fall gibt es noch die Sequenz von weiteren Programmen, die in einer Liste gespeichert sind. Wir iterieren durch diese Liste und rufen für jedes Teilprogramm un dieser Liste die Methode rekursiv auf.

Java: Expr

```
case Seq el -> {for (var e:el.es) e.collectVars(vs);}
```

Ein einziger Fall wurde noch nicht berücksichtigt, der Fall dass das Programm eine einzelne Zahl ist. Hier ist nichts zu tun, denn es ist damit ja keine weitere Variable im Programm enthalten. Da es sich bei `Expr` um eine versiegelte Schnittstelle handelt, prüft der Javacompiler, ob im `switch-case`-Ausdruck auch alle Fälle berücksichtigt sind. Wir dürfen also diesen siebten Fall nicht weglassen.

Java: Expr

```
    case Num ne -> {}
  }
}
```

Die so geschriebene Methode lässt sich zur einfachen Verwendung noch kapseln.

Java: Expr

```
default Set<String> collectVars(){
  Set<String> result = new HashSet<String>();
  collectVars(result);
  return result;
}
```

Übungsaufgabe 4.1. *Jetzt sind Sie dran, eine default-Methode für die Schnittstelle zu schreiben. Diese soll das Programm ausführen und über die Variablebelegung in einem* `Map` *Buch führen.*

Das Ergebnis der Methode soll dabei der Wert des ausgeführten Programms sein. Im Falle der Anweisungen, also der if-Verzweigung, der Scheife ist der Rückgabewert irrelevant. Bei Sequenz von Teilprogrammen sei es der Wert des letzten Teilprogramms.

Die ersten zwei Fälle sind bereits ausgeführt. Bei einer Variablen wird der Wert der Variablen in dem `Map`*-Objekt* `env` *nachgeschlagen. Bei einer Zahlenkonstante ist das Ergebnis, die dargestellte Zahl.*

Im Fall der Variablen sehen wir noch ein weiteres Java Schlüsselwort: `yield`*. Dieses erfüllt für den* `switch-case`*-Ausdruck ungefähr die Funktion, die ein* `return` *in Methoden hat. Es soll das Ergebnis des Ausdrucks, also des* `switch-case` *damit angegeben werden. Hier benötigen wir es, weil es in dem* `case Var` *erst noch eine Printanweisung gibt. Java wird mit dem Schlüsselwort* `yield` *jetzt mitgeteilt, dass nun das eigentliche Ergebnis für diesen* `case` *folgt.*

Java: Expr

```
default long eval(Map<String,Long> env){
  return switch(this){
    case Num num -> num.n;
    case Var v -> {
      System.out.println("looking for variable: "+v.name);
      yield env.get(v.name);
    }
    default -> 0L; //TODO add all missing cases
  };
}
```

4.1.1. Ausführung eines Programmtextes

Wenn Sie die Aufgabe gelöst haben, führt folgende Methode den Programmtext eines Skriptes der kleinen Skriptsprache aus und gibt die abschließende Variablenbelegung als Ergebnis zurück.

Java: Expr

```
static Map<String, Long> eval(String skript) {
  var result = new HashMap<String,Long>();
  eval(new StringReader(skript),result);
  return result;
}
```

Die eigentliche Methode zur Ausführung der Skripte muss zunächst den Programmtext in die Klassen dieser Aufgabe umwandeln. Hierzu wird der im Anhang implementierte Parser verwendet.

Java: Expr

```
static void eval(Reader skript, Map<String, Long> env) {
  try {
    var lexer = new SimpleScriptLexer(CharStreams.fromReader(skript));
    var parser = new SimpleScriptParser(new CommonTokenStream(lexer));

    var antlrtree = parser.script();
    var tree = new BuildTree().visit(antlrtree);
    tree.eval(env);
  } catch (Exception e) {
    System.out.println(e);
  }
}
```

4.2. Hauptmethode zum Ausführen von Skriptdateien

Mit der folgenden kleinen Hauotmethode lassen sich Skriptdateien direkt ausführen:

Java: Expr

```
  public static void main(String... args){
     for (var arg:args){
       try {
         var env = new HashMap<String,Long>();
         eval(new java.io.FileReader(arg),env);
         System.out.println(env);
       }catch(Exception e){
         e.printStackTrace();
       }
     }
  }
}
```

5. Beispielsession

Wir können die Skriptsprache direkt in der JShell einmal ausprobieren. Hierzu ist notwendig, den Klassenpfad so zu setzen, dass er auf die jar-Datei von ANTLR verweist und den Ordner, in dem die class-Dateien kompilietr vorliegen.

Shell

```
px1:~$ jshell --enable-preview --class-path
↪  classes/:antlr-4.9.1-complete.jar
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro
```

Wir können in der JShell-Session die gewünschte Klassen importieren:

Shell

```
jshell> import name.panitz.util.Expr

jshell> import static name.panitz.util.Expr.*
```

Als erstes sei einmal direkt mit den Recordklassen das Skript zur Berechnung der Fakultät von 5 definiert.

Shell

```
jshell> var fac = new Seq(
   ...>   List.of
   ...>    ( new AssignEx("r",new Num(1))
   ...>    , new AssignEx("x",new Num(5))
   ...>    , new WhileEx(new Var("x"),new Seq(List.of
   ...>      ( new AssignEx("r",new OpEx(new Var("r"),Op.mult,new
   ↪  Var("x")))
   ...>      , new AssignEx("x",new OpEx(new Var("x"),Op.sub,new Num(1)))
   ...>      )
   ...>     ))
   ...>    , new Var("r")
   ...>    )
   ...> );
fac ==> Seq[es=[AssignEx[v=r, e=Num[n=1]], AssignEx[v=x,  ...
↪  m[n=1]]]]]], Var[name=r]]]
```

Auf diesem Programm kann direkt die Methode zur Berechnung aller dort vorkommenden Variablen aufgerufen werden.

Shell

```
jshell> fac.collectVars()
$4 ==> [r, x]
```

Das Programm kann auch für eine initial leere Belegung der Variablen ausgeführt werden:

Shell

```
jshell> fac.eval(new HashMap<>())
$5 ==> 120
```

Wir können auch Skripte direkt als String zur Ausführung bringen.

Shell

```
jshell> Expr.eval("x=17+4*2")
$6 ==> {x=25}
```

Schließlich noch ein Beispiel mit zwei Variablen, denen ein Wert zugewiesen wird.

Shell

```
jshell> Expr.eval(" {x=17+4*2 y=2*x}")
$7 ==> {x=25, y=50}

jshell>
```

A. Grammatik

In diesem Kapitel wird für das Compilerbautool ANTLR die Grammatik der kleinen Skrpitsprache definiert.

A.1. Paketdeklaration im Kopf

Die Sprache heiße `SimpleScript` und liegt im Paket `name.panitz.util`.

Java: SimpleScript

```
grammar SimpleScript;

@header {
package name.panitz.util;
}
```

A.2. Regeln der Grammatik

Die Grammatik wird als kontextfreie Regel dargestellt.

Java: SimpleScript

```
script
  : script (MULT | DIV| MOD) script
  | script  (ADD | SUB) script
  | VARIABLE  EQ script
  | VARIABLE
  | NUMBER
  | IF script THEN script ELSE script
  | WHILE script DO script
  | LBRACE (script)+ RBRACE
  | LPAREN script RPAREN
;
```

A.3. Schlüsselwörter

Die Schlüsselwörter der Sprach sind hier aufgelistet.

Java: SimpleScript

```
WHILE: 'while';
DO: 'do';
IF: 'if';
THEN: 'then';
ELSE: 'else';
```

A.4. Bezeichner

Ein Bezeichner (Variablenname) beginnt mit einem Buchstaben gefolgt von einer Buchstabenzahlenfolge.

Java: SimpleScript

```
VARIABLE
   : VALID_ID_START VALID_ID_CHAR*
   ;

fragment VALID_ID_START: ('a' .. 'z') | ('A' .. 'Z') | '_';
fragment VALID_ID_CHAR: VALID_ID_START | ('0' .. '9') ;
```

A.5. Zahlenliterale

Eine Zahl ist eine Folge von Ziffern:

Java: SimpleScript

```
NUMBER: ('1' .. '9')('0' .. '9')* | '0';
```

A.6. Symbole

Folgende Symbole werden in der Sprache verwendet.

Java: SimpleScript

```
LPAREN: '(';
RPAREN: ')';
LBRACE: '{';
RBRACE: '}';
SEMICOLON: ';';
ADD: '+';
SUB: '-';
MULT: '*';
DIV: '/';
MOD: '%';
EQ: '=';
```

A.7. Kommentare und Zwischenraum

Der übliche Zwischenraum ist abschließend definiert:

Java: SimpleScript

```
LINE_COMMENT: '//' .*?  '\n' -> skip; //Match "//" stuff '\n'
COMMENT: '/*' .*? '*/' -> skip;  //Match /* stuff */

WS : [ \r\n\t] + -> skip;
```

B. Abstrakter Syntaxbaum

Folgende Klasse baut aus den automatisch generierten Klassen des Tools ANTLR ein Objekt unserer Schnittstelle `Expr`.

Java: BuildTree

```java
package name.panitz.util;
import org.antlr.v4.runtime.tree.AbstractParseTreeVisitor;
import static name.panitz.util.Expr.*;
import java.util.stream.Collectors;

public class BuildTree
    extends AbstractParseTreeVisitor<Expr>
    implements SimpleScriptVisitor<Expr> {
  @Override
  public Expr visitScript(SimpleScriptParser.ScriptContext ctx) {
    if (ctx.WHILE() != null) {
      return  new WhileEx(visit(ctx.script(0)), visit(ctx.script(1)));
    }else if (ctx.IF() != null) {
      return  new IfEx( visit(ctx.script(0))
                      , visit(ctx.script(1))
                      , visit(ctx.script(2)));
    }else if (ctx.LPAREN() != null) {
      return  visit(ctx.script(0));
    }else if (ctx.LBRACE() != null) {
      return new Seq(ctx.script()
                        .stream()
                        .map(s->visit(s))
                        .collect(Collectors.toList()));
    }else if (ctx.EQ() != null) {
      return new AssignEx(ctx.VARIABLE().getText(),visit(ctx.script(0)));
    }else if (ctx.VARIABLE() != null) {
      return new Var(ctx.VARIABLE().getText());
    }else if (ctx.NUMBER() != null) {
      return new Num(Long.parseLong(ctx.NUMBER().getText()));
    } else{
      var op = Op.add;
      if (ctx.SUB() != null) {
        op = Op.sub;
      }else if (ctx.DIV() != null) {
        op = Op.div;
      }else if (ctx.MULT() != null) {
        op = Op.mult;
      }else if (ctx.MOD() != null) {
        op = Op.mod;
      }
      return new OpEx(visit(ctx.script(0)),op,visit(ctx.script(1)));
    }
  }
}
```