Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Compiler und Interpreter

– Praktikum, Teil 4 –

Prof. Dr. Michael Neitzke

„Zielcode“ mittels String Templates ausgeben

- Sie haben jetzt einen Mechanismus entwickelt, um ein Symbolrätsel in sechs Symbol-Additionsaufgaben zu überführen.
- Außerdem haben Sie mit Choco ein Programm zum Constraint-basierten Lösen von Symbol-Additionsaufgaben entwickelt.
- Sie können nun auch in derselben Weise die sechs Symbol-Additionsaufgaben mit Choco umsetzen. Es handelt sich hierbei ja lediglich um ein größeres, eben sechs Aufgaben umfassendes Constraint-Netz.
- Dieses Programm soll aber nicht per Hand, sondern als Ausgabe eines ANTLR-Übersetzungsprozesses erzeugt werden, und zwar mit Hilfe von String Templates. Einige Bestandteile des Programms sind ja, unabhängig von der konkreten Symbolaufgabe immer gleich, andere von der konkreten Symbolaufgabe abhängig. String Templates sind genau dafür da, eine Ausgabe zu erzeugen, die abhängig von der konkreten Eingabe variable Bausteine enthält.

Anleitung

- Im Folgenden finden Sie Lösungsbestandteile. Hier können Sie Anregungen für Ihre eigene Lösung gewinnen. Das hier vorgestellt Programm schreibt jedoch nur zu addierende Symbole heraus. Die Struktur kann jedoch für die Praktikumsaufgabe verwendet werden. Am Schluss folgt eine Anleitung darüber, welche Veränderungen erforderlich sind.

Lösungsbestandteile: Main Methode

```
CommonTree r2 = ((CommonTree) ast2.tree);
System.out.println("\n\nnach dem normalisieren");
System.out.println(r2.toStringTree());


CommonTreeNodeStream nodes2 = new CommonTreeNodeStream(r2);
nodes2.setTokenStream(tokens);
SymbolraetselEmitter emitter = new SymbolraetselEmitter(nodes2);
InputStream templateIs = Main.class.getClassLoader()
      .getResourceAsStream(TEMPLATE_FILE);

StringTemplateGroup templates = new StringTemplateGroup(
      new InputStreamReader(templateIs, "ISO-8859-15"),
      AngleBracketTemplateLexer.class);
emitter.setTemplateLib(templates);
SymbolraetselEmitter.puzzle_return puzzle_return = emitter.puzzle();
String output = puzzle_return.getTemplate().toString();
System.out.println("\n\nausgabe");
System.out.println(output);
```

Lösungsbestandteile: Klasse Number

```
public class Number {
    char[] digits;
    int maxSize;

    public void setDigits(List<Tree> tokens) {
        digits = new char[tokens.size()];
        for (int i = 0; i < tokens.size(); i++) {
            Tree tree = tokens.get(i);
            char c = tree.getText().charAt(0);
            digits[i] = c;}
    }

    public List<Character> getCharacters() {
        List<Character> number = new ArrayList<Character>();
        for (int i = maxSize - digits.length; i > 0; i--) {
            number.add(null);}
        for (int i = 0; i < digits.length; i++) {
            number.add(digits[i]);}
        return number;
    }

    public int getSize() {
        return digits.length;
    }
}
```

Lösungsbestandteile: Klasse Constraint

```
public class Constraint {
    List<Number> numbers = new ArrayList<Number>();

    public void prepare() {
        int maxSize = -1;
        for (Number number : numbers) {
            if (number.getSize() > maxSize) {
                maxSize = number.getSize();
            }
        }
        for (Number number : numbers) {
            number.maxSize = maxSize;
        }
    }
}
```


Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Lösungsbestandteile: Emitter (1)

```
tree grammar SymbolraetselEmitter;

options {
  tokenVocab   = Symbolraetsel;
  output       = template;
  ASTLabelType = CommonTree;
}

puzzle
  :
  ^(PUZZLE constraints+=constraint*)
  -> sums(sums={$constraints})
  ;
```

Lösungsbestandteile: Emitter (2)

```
constraint
@after {
Constraint constraint = new Constraint();
constraint.numbers.add($n1.number);
constraint.numbers.add($n2.number);
constraint.numbers.add($n3.number);
constraint.prepare();
}
  :
  ^(
    EQ
    ^(PLUS n1=number n2=number)
    n3=number
   )
   -> sum(number1={$n1.number}, number2={$n2.number},
number3={$n3.number})
  ;
```

Lösungsbestandteile: Emitter (3)

```
number returns [Number number]
@after {
$number = new Number();
$number.setDigits($syms);
}
  :
  ^(NUMBER syms+=SYM+)
  ;
```

String-Template

```
group template;

sums(symbols,sums) ::= <<
<sums; separator="\n\n">
>>

sum(number1, number2, number3) ::= <<
<number1.characters, number2.characters, number3.characters :
  {n1, n2, n3 | <if(n1)><n1><else>0<endif> + <if(n2)><n2><else>0<endif> =
<if(n3)><n3><else>0<endif>;}
  ; separator="\n">
>>
```

Was muss an dieser Lösung verändert werden?

- Der Emitter muss die Symbole sammeln und dem Template „sums" übergeben.
- Ein Template analog zu „sums" schreibt den gesamten Choco-Code bis auf die Definition der Übertrags-Variablen und der Additions-Constraints.
  - Es gibt statische Anteile wie z. B. die Import-Statements oder das Erzeugen eines Modells.
    - Wie man durch eine Liste durchiteriert, kann man am Template „sum" sehen.
  - Es gibt dynamische Anteile, nämlich die Definitionen für die Symbol-Variablen und den All-Different-Constraint
- Ein Template analog zu „sum" erzeugt die Definitionen für die Übertrags-Variablen und die Additions-Constraints.
  - Die Bezeichner für die Übertrags-Variablen müssen systematisch erzeugt werden, da es ja sechs Additions-Aufgaben mit jeweils mehreren Übertrags-Variablen gibt.
  - Da die Zahlen beliebig viele Symbole enthalten dürfen, müssen in den Additions-Constraints manchmal Nullen statt eines Symbols eingegeben werden. Hierbei kann man sich am Template „sum" orientieren.