PROGRAMMIERMETHODEN: Hinweise zum 6. Aufgabenblatt

FH Bielefeld
University of
Applied Sciences

SS 2012
Prof. Dr.-Ing. C. Gips

RegExps, Logging, Testen, Versionierung

Abgabe: KW24 (11. – 15. Juni 2012)

H I N W E I S E
zu Blatt 06

In Aufgabe 1 von Blatt 06 soll mit Hilfe von regulären Ausdrücken ein einfacher Lexer programmiert werden. Bitte beachten Sie folgende Hinweise und Korrekturen zur Aufgabenstellung.

Hinweise:

- Der Text eines Kommentares gehört zum jeweiligen Kommentar-Token!
- Sie können bei Kommentar-Token die Kommentarmarkierer (Start- und eventuelle Endezeichen) mit in den Tokeninhalt speichern oder auch nicht – durch den Tokentyp ist an sich klar, um welchen Kommentarmarkierer es sich handelt.
  Beispiel:

```
// total irrer Kommentar
```

  enthält das "einzeiliger Kommentar"-Token mit dem Inhalt `total irrer Kommentar` bzw. `// total irrer Kommentar`

- Das "sonstiger Text"-Token soll jeweils den Text zwischen zwei Token beinhalten, der selbst kein Token ist. Leerzeichen zwischen zwei Token werden dabei ignoriert.
  Beispiel:

```
public void myMethod(String s) {}
// total irrer Kommentar
```

  sollte die Schlüsselwort-Token `public`, `void` und das "sonstiger Text"-Token `myMethod(String s) {}` liefern, gefolgt vom "Zeilenumbruch"-Token (s.u.) und dem "einzeiliger Kommentar"-Token `total irrer Kommentar`

- Nach einem erfolgreichen Durchlauf ist der Stream leer und es ergibt sich eine Folge von Token. Wenn man die String-Repräsentation der Token in dieser Reihenfolge verkettet (jeweils mit Leerzeichen getrennt), sollte sich wieder der ursprüngliche Text ergeben (plus/minus ein paar zusätzliche, für den Java-Quellcode nicht relevante Leerzeichen).

Korrektur/Präzisierung von Aufgabe 1:

- Nutzen Sie zur Darstellung von Zeilenumbrüchen ein zusätzliches spezielles "Zeilenumbruch"-Token. Diese Tokenklasse hat eine höhere Priorität als das "sonstiger Text"-Token, muss also vor diesem geprüft werden.
  Zeilenumbrüche innerhalb von mehrzeiligen Kommentaren gehören aber nach wie vor zum Kommentartext!
- Der Umgang mit dem "sonstiger Text"-Token ist etwas schwierig und aus softwaretechnischer Sicht unschön, da alle (anderen) Tokenklassen unabhängig voneinander sind und nur diese eine Tokenklasse die restlichen Tokenklassen kennen muss (zumindest deren reguläre Ausdrücke).

Sie können dieses Problem leicht durch ein zweistufiges Verfahren umgehen.

1. Die "sonstiger Text"-Klasse enthält einen regulären Ausdruck, der auf genau ein beliebiges Zeichen matcht. Bei einem Durchlauf wird diese Tokenklasse wie gehabt als letzte Tokenklasse geprüft, d.h. wenn es in einem Durchlauf kein "echtes" Token gibt, erhalten Sie eine Instanz des "sonstiger Text"-Tokens, welches genau ein Zeichen vom Streamanfang entfernt hat. Danach starten Sie wie gehabt den nächsten Durchlauf und wiederholen dieses Verfahren, bis der Stream leer ist. Im Anschluss haben Sie eine Folge von Token, die Ihren Text repräsentieren.
2. In der resultierenden Folge von Token werden benachbarte "sonstiger Text"-Token zu einem "sonstiger Text"-Token zusammengefasst.

Beispiel:

```
/* Kommentar
   eins */
public void myMethod(String s) {}
// total irrer Kommentar
```

sollte zunächst die Tokenfolge `Kommentar\n  eins`, "Zeilenumbruch"-Token, `public`, `void`, `m`, `y`, `M`, ..., `}`, "Zeilenumbruch"-Token, `total irrer Kommentar` finden. Nach dem Zusammenfassen benachbarter "sonstiger Text"-Token ergibt sich die Folge `Kommentar\n  eins`, "Zeilenumbruch"-Token, `public`, `void`, `myMethod(String s) {}`, "Zeilenumbruch"-Token, `total irrer Kommentar`, die entsprechend weiter verarbeitet werden kann (Rückgabewert o.ä.).

Frage: Wieviele Token ergeben sich bei folgendem Beispiel und warum?

```
/* Kommentar
   eins
public void myMethod(String s) {}
// total irrer Kommentar
```