Aufgabe 5: Extraktion von Dokumentationskommentaren

Für Java gibt es ein Werkzeug *javadoc*, das aus einem Java-Quellprogramm Dokumentationskommentare extrahiert. Wir wollen ein ähnliches Werkzeug für MicroJava implementieren. Ein Dokumentationskommentar beginnt mit /** und endet mit */. Dazwischen dürfen beliebige Texte stehen, aber der Kommentar darf nicht geschachtelt werden. In MicroJava sollen Dokumentationskommentare vor die Deklaration von Konstanten, Variablen, Klassen und Methoden gesetzt werden können. Im Kommentar können Tags vorkommen, von denen wir hier nur zwei implementieren, und zwar für Dokumentationskommentare, die vor Methodendeklarationen stehen:

```
@param ident ... Text ...
@returns ... Text ...
```

Das @param-Tag beschreibt einen Parameter (namens ident), das @returns-Tag die Bedeutung des Rückgabewerts einer Methode. Die Texte gehen bis zum nächsten Tag oder bis zum Ende des Dokumentationskommentars.

Implementieren Sie mittels Coco/R ein Werkzeug, das aus einem MicroJava-Programm alle Dokumentationskommentare extrahiert und zusammen mit dem Namen des so kommentierten Sprachelements (Konstante, Variable, Klasse oder Methode) ausgibt. Tags und ihre Bestandteile sollen dabei ebenfalls gelistet werden.

Sie brauchen nur die relevanten Teile eines MicroJava-Programms durch eine Grammatik beschreiben. Irrelevante Teile können durch {ANY} überlesen werden (»Fuzzy-Parsing«, siehe die Beschreibung des Symbols ANY in Kapitel 7.2).

Lösung

Wir verwalten die Teile eines Dokumentationskommentars in einer Klasse Comment

```
class Comment {
   ArrayList<Element> elements = new ArrayList<Element>();
   void add (Element e) { elements.add(e); }
}
```

die ein Array von Elementen enthält, die in diesem Kommentar vorkommen:

```
class Element {
   String kind;   // tag name or null
   String name; // ident to which the tag refers or null
   String text;   // comment text
}
```

Für den Dokumentationskommentar

```
/** Prints a value and starts a new line after 10 values
@param x the value to be printed */
void print (int x) { ... }
```

werden zwei Elemente angelegt:

```
kind == null, name == null, text == "Prints a value and starts a new line after 10 values"
kind == "@param", name == "x", text == "the value to be printed"
```

Um die Texte aus dem Quellprogramm zu extrahieren, lesen wir das gesamte Quellprogramm in einen Puffer, der wie folgt deklariert wird:

```
public class Buffer {
   byte[] source = new byte[10000];  // source text

   String range (int beg, int end) {  // extracts a range from the buffer
      while (source[beg] <= ' ') beg++;
      while (source[end] <= ' ') end--;
      return new String(source, beg, end - beg + 1);
   }
```

```
  Buffer (FileInputStream s) {  // constructor; reads the source file into the buffer
    try {
      s.read(source);
    } catch (IOException e) {
      System.out.println("-- IO error");
    }
  }
}

Buffer buf;  // the source code buffer
```

In der Scannerbeschreibung deklarieren wir die Terminalsymbole von MicroJava und geben an, dass Zeilenenden und Tabulatoren ignoriert werden sollen.

In der Parserbeschreibung legen wir eine neue Produktion DocComment für Dokumentationskommentare an und beschreiben dort ihre Syntax. Alle Teile außer den Kommentarklammern, den Tag-Namen und dem ident beim @param-Tag überlesen wir mittels {ANY}. Wir merken uns die Anfangs- und Endposition des Kommentars und kopieren diesen Bereich aus dem Quelltext-Puffer.

Optionale Dokumentationskommentare werden anschließend vor die einzelnen Arten von Deklarationen gesetzt.

Hier ist die entsprechende Compilerbeschreibung:

```
import java.util.ArrayList;
import java.io.*;

COMPILER Doc

//------------------------------ global declarations ------------------------------
  public class Buffer {
    byte[] source = new byte[10000];  // source text

    String range (int beg, int end) { // extracts a range from the buffer
      while (source[beg] <= ' ') beg++;
      while (source[end] <= ' ') end--;
      return new String(source, beg, end - beg + 1);
    }

    Buffer (FileInputStream s) { // constructor; reads the source file into the buffer
      try {
        s.read(source);
      } catch (IOException e) {
        System.out.println("-- IO error");
      }
    }
  }

  class Element {
    String kind;    // tag name or null
    String name;    // ident to which the tag refers or null
    String text;    // comment text
  }

  class Comment {
    ArrayList<Element> elements = new ArrayList<Element>();
    void add (Element e) { elements.add(e); }
  }

  public Buffer buf;
```

```
   void print (String elemKind, String name, Comment com) {  // prints the collected data about comments
      System.out.println("comment for " + elemKind + " " + name);
      for (Element e: com.elements) {
         System.out.print("  ");
         if (e.kind != null) System.out.print(e.kind + " ");
         if (e.name != null) System.out.print(e.name + " ");
         System.out.println(e.text);
      }
   }

//------------------------------ scanner specification ------------------------------
CHARACTERS
  letter  = 'A'..'Z' + 'a'..'z'.
  digit   = '0'..'9'.
  noQuote = ANY - '\''.

TOKENS
  ident     = letter {letter | digit | '_'}.
  number    = digit {digit}.
  charConst = '\'' (noQuote | '\\' ('r' | 'n' | 't')) '\''.

COMMENTS FROM "//" TO "\r\n"

IGNORE '\r' + '\n' + '\t'

//------------------------------- parser specification -------------------------------
PRODUCTIONS

Doc
= "program" ident          (. Comment com = null; .)
   {   [ DocComment <out com> ]
      ( ConstDecl <com>
      | VarDecl <com>
      | ClassDecl <com>
      )
   } "{" {MethodDecl} "}".

ConstDecl <Comment com>
=  "final" Type ident       (. if (com != null) print("constant", t.val, com); .)
   {ANY} ";".

VarDecl <Comment com>
=  Type ident               (. if (com != null) print("variable", t.val, com); .)
   {ANY} ";".

ClassDecl <Comment com>
=  "class" ident            (. if (com != null) print("class", t.val, com); .)
   "{" {ANY} "}".

MethodDecl                  (. Comment com = null; .)
=  [ DocComment <out com> ]
   (Type | "void") ident    (. if (com != null) print("method", t.val, com); .)
   {ANY}
   Block .

Block
= "{"
   {  Block
   |  ANY
   }
   "}".

Type   = ident ["[" "]"].
```

```
DocComment <out Comment com>
= "/**"                    (. com = new Comment();
                              Element e = new Element();
                              int beg = la.pos; .)
  {ANY}                    (. e.kind = null;
                              e.name = null;
                              e.text = buf.range(beg, la.pos - 1);
                              com.add(e); .)
  {  "@param" ident        (. e = new Element();
                              e.kind = "@param";
                              e.name = t.val;
                              beg = la.pos; .)
     {ANY}                 (. e.text = buf.range(beg, la.pos - 1);
                              com.add(e); .)
  |  "@returns"            (. e = new Element();
                              e.kind = "@returns";
                              e.name = null;
                              beg = la.pos; .)
     {ANY}                 (. e.text = buf.range(beg, la.pos - 1);
                              com.add(e); .)
  }
  "*/".

END Doc.
```

Ein Dokumentationskommentar enthält nach der öffnenden Klammer /** beliebigen Text bis zum Beginn des ersten Tags oder bis zum Ende des Kommentars. Dieser Text wird mit {ANY} überlesen. ANY bedeutet hier jedes Terminalsymbol, das keine Alternative zu diesem ANY ist, also alles außer @param, @returns und */. Nach dem @param-Tag wird der Name des Parameters (ident) erkannt, und anschließend wird wieder alles bis zum nächsten Tag oder bis zum Ende des Kommentars mit {ANY} überlesen. Ähnliches gilt für das @returns-Tag.

Vor jedem {ANY} holen wir uns mit la.pos die Quelltextposition des nächsten Symbols, und nach dem {ANY} zeigt la.pos wieder die Position des nächsten Symbols an, daher müssen wir die Position um 1 verringern. Wir kopieren dann den entsprechenden Bereich aus dem Quelltext-Puffer und speichern ihn in e.text.

Vor jede Deklaration einer Konstanten, Variablen, Klasse oder Methode setzen wir einen optionalen Dokumentationskommentar und gegen dann zur Kontrolle den deklarierten Namen und den dazugehörigen Dokumentationskommentar mit der Methode print aus.

Wir brauchen noch ein Hauptprogramm, das den Namen der Eingabedatei als Kommandozeilenparameter liest, einen Scanner und einen Parser erzeugt und die Syntaxanalyse startet. Vorher wird aber noch der Quelltext-Puffer gefüllt.

```
import java.io.*;

class Doc {

   public static void main (String[] arg) {
      if (arg.length > 0) {
         Scanner scanner = new Scanner(arg[0]);
         Parser parser = new Parser(scanner);
         try {
            FileInputStream s = new FileInputStream(arg[0]);
            parser.buf = parser.new Buffer(s);  // fill source code buffer
            parser.Parse();
            System.out.println(parser.errors.count + " errors detected");
         } catch (FileNotFoundException e) {
            System.out.println("-- file " + arg[0] + " not found");
         }
      } else
         System.out.println("-- No source file specified");
   }
}
```

Wir verarbeiten die Compilerbeschreibung mit Coco/R:

```
java -jar Coco.jar Doc.atg
```

wodurch ein Scanner und ein Parser erzeugt wird. Anschließend übersetzen wir alle Java-Dateien

```
javac Scanner.java Parser.java Doc.java
```

und rufen dann das fertige Werkzeug wie folgt für ein MicroJava-Beispielprogramm (z.B. QuickSort.mj) auf:

```
java Doc QuickSort.mj
```

Wenn QuickSort.mj folgendes MicroJava-Programm mit Dokumentationskommentaren enthält

```
program QuickSort
   /** The size of the array to be sorted */
   final int SIZE = 20;

   /** The array to be sorted */
   int[] data;

   /** A seed for the random number generator */
   int seed;

{  /** Implementation of Quicksort
   @param data the array to be sorted
   @param beg  the lower index of the range to be sorted
   @param end  the upper index of the range to be sorted */
   void sort (int[] data, int beg, int end)
      int i, j, m, x;
   {  if (beg < end) {
         i = beg; j = end;
         m = data[(i + j) / 2];
         while (i <= j) {
            while (data[i] < m) i++;
            while (data[j] > m) j--;
            if (i > j) break;
            if (i != j) { x = data[i]; data[i] = data[j]; data[j] = x; }
            i++; j--;
         }
         sort(data, beg, j);
         sort(data, i, end);
      }
   }

   /** Random number generator
   @returns a random number in the range 0..99 */
   int randNum() {
      seed = seed * 1103515245;
      return 50 + seed % 50;
   }

   /** The main program */
   void main()
      int i;
   {
      data = new int[SIZE];
      seed = 123456789;
      i = 0;
      while (i < SIZE) {
         data[i] = randNum(); i++;
      }
      sort(data, 0, SIZE-1);
      i = 0;
      while (i < SIZE) {
         print(data[i], 3); i++;
      }
   }
}
```

liefert der Aufruf von Doc folgendes Ergebnis:

```
comment for constant SIZE
   The size of the array to be sorted
comment for variable data
   The array to be sorted
comment for variable seed
   A seed for the random number generator
comment for method sort
   Implementation of Quicksort
   @param data the array to be sorted
   @param beg the lower index of the range to be sorted
   @param end the upper index of the range to be sorted
comment for method randNum
   Random number generator
   @returns a random number in the range 0..99
comment for method main
   The main program
```