Exercise 5: Extraction of Documentation Comments

For Java, there is a tool *javadoc* that extracts documentation comments from a Java source program. We want to implement a similar tool for MicroJava. A documentation comment starts with /** and ends with */. Any text may be placed in between, but the comments must not be nested. In MicroJava, it should be possible to place documentation comments in front of the declaration of constants, variables, classes, and methods. Documentation comments can contain *tags*, of which we only implement two here, namely for comments that precede method declarations:

```
@param ident ... text ...
@returns ... text ...
```

The @param tag describes a parameter with the name ident, and the @returns tag describes the method's return value. The texts, which explain the parameter or return value, continue until the next tag or until the end of the comment.

Use Coco/R to implement a tool that extracts all documentation comments from a MicroJava program and outputs them. For each commented program element (constant, variable, class, or method), list its name as well as its documentation comment including any tags and their components.

In your grammar, you only need to specify the relevant parts of a MicroJava program. Irrelevant parts can be skipped by {ANY} (*fuzzy parsing*, see Section 7.2).

Solution

We manage the parts of a documentation comment in a class Comment

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

which contains an array of the elements that occur in a comment:

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

For example, for the documentation comment

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

the following two elements are created:

```
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"
```

To extract the texts from the source program, we read the entire source program into a buffer that is declared as follows:

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

  String range (int beg, int end) {  // extracts the range [beg..end] 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 the scanner specification, we declare MicroJava's terminal symbols and specify that line endings and tabs should be ignored.

In the parser specification, we introduce a new production DocComment for documentation comments and describe its syntax there. We skip all parts except the comment brackets, the tag names and the ident in the @param tag using {ANY} (*fuzzy parsing*) We take the start and the end position of the comment and copy this area from the source code buffer.

In front of the declarations of constants, variables, classes and methods we allow for optional documentation comments, leading to the following compiler specification:

```
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.
```

A documentation comment contains any text after the opening bracket /** up to the beginning of the first tag or up to the end of the comment. This text is skipped with {ANY}. ANY here means any terminal symbol that is not an alternative of this ANY, i.e., everything except @param, @returns and */. After the @param tag, the name of the parameter (ident) is recognized, and then everything up to the next tag or up to the end of the comment is skipped again with {ANY}. The same applies to the @returns tag

Before each occurrence of {ANY}, we retrieve the source text position of the next symbol with la.pos. After each {ANY}, la.pos contains the position of the next symbol, so we have to decrement this position by 1. We then copy the corresponding range from the source text buffer and save it in e.text.

In front of each declaration of a constant, variable, class or method, we place an optional documentation comment. In the corresponding semantic action we print the declared name and the contents of the documentation comment with the print() method for checking purposes.

We still need a main program that gets the name of the input file as a command line argument, generates a scanner and a parser and starts the syntax analysis. Before starting the parser, however, the source code buffer has to be filled.

```
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");
  }
}
```

We process the compiler specification with Coco/R:

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

thus generating a scanner and a parser. Then we compile all Java files

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

and invoke our tool for a specific MicroJava program (e.g. QuickSort.mj):

```
java Doc QuickSort.mj
```

If QuickSort.mj contains the following MicroJava program with documentation comments

```
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++;
    }
  }
}
```

the invocation of Doc leads to the following result:

```
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
```