Aufgabe 4: Komplexitätsanalyse

Um die Komplexität eines Programms zu ermitteln, werden oft aus seinem Quelltext Maßzahlen errechnet, die eine Einschätzung erlauben, wie schwer ein Programm zu verstehen und zu warten ist. Es gibt verschiedene Komplexitätsmaße, aber wir wollen hier eines definieren, das die Komplexität einzelner MicroJava-Methoden aus ihren Anweisungen berechnet. Dazu legen wir für jede Anweisungsart ihre Grundkomplexität fest:

- Zuweisung 1
- Methodenaufruf 2
- ++ oder -- 1
- if-Anweisung 1 + Komplexität der geschachtelten Anweisungen
- while-Anweisung 1 + Komplexität der geschachtelten Anweisungen
- break-Anweisung 5
- return-Anweisung 1
- read-Anweisung 1
- print-Anweisung 1

Die Komplexität einer Methode ist die Summe der Komplexitäten ihrer Anweisungen. Da Schachtelungen die Komplexität erhöhen, wird die Komplexität geschachtelter Anweisungen mit dem Faktor 2 multipliziert. Die Anweisung

```
if (x > 0) val = 1; else val = 2;
```

hat zum Beispiel die Gesamtkomplexität 5: Jede Zuweisung hat die Grundkomplexität 1, die hier mit dem Faktor 2 multipliziert wird, was insgesamt 4 ergibt; dazu kommt die Grundkomplexität 1 der if-Anweisung. Wenn die if-Anweisung selbst eingeschachtelt ist, wird ihre Gesamtkomplexität wieder mit dem Faktor 2 multipliziert, usw.

Erzeugen Sie mittels Coco/R ein Programm, das die Komplexität aller darin enthaltenen Methoden berechnet und ausgibt. Verwenden Sie als Parserbeschreibung die MicroJava-Grammatik aus Anhang A und attributieren Sie sie so, dass die Komplexität von Methoden aus den Grundkomplexitäten ihrer Anweisungen berechnet wird. Funktionsaufrufe in Ausdrücken können Sie dabei ignorieren.

Lösung

Als Terminalsymbole haben wir ident, number und charConst, die in der Scannerspezifikation deklariert werden. Neben Leerzeichen sollen Zeilenenden und Tabulatoren ignoriert werden.

Als Parserbeschreibung verwenden wir die Grammatik von MicroJava aus Anhang A. Die meisten Produktionen erfordern keine Attributierung. Nur bei Methodendeklarationen, Statements und Anweisungsblöcken müssen wir etwas tun.

```
COMPILER Metrics

//------------------ global semantic declarations -----------------
   static int factor = 2; // increased complexity factor for nested statements

//----------------------- 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

Metrics      = "program" ident {ConstDecl | VarDecl | ClassDecl} "{" {MethodDecl} "}".
ConstDecl = "final" Type ident "=" (number | charConst) ";".
VarDecl      = Type ident {"," ident } ";".
ClassDecl  = "class" ident "{" {VarDecl} "}".

MethodDecl                      (. int complexity; .)
=  (Type | "void") ident         (. String name = t.val; .)
   "(" [FormPars] ")"
   {VarDecl}
   Block <out complexity>       (. System.out.println(name + ": " + complexity); .) .

Block <out int complexity>      (. int c; .)
=  "{"                          (. complexity = 0; .)
   { Statement <out c>          (. complexity += c; .)
   }
   "}".

Statement <out int complexity>  (. int c; .)
=                               (. complexity = 1; .) // default
   (  Designator  ( "=" Expr
                  | ActPars     (. complexity = 2; .)
                  | "++"
                  | "--"
                  ) ";"
   |  "if" "(" Condition ")"
      Statement <out c>         (. complexity += c * factor; .)
      [  "else"
         Statement <out c>      (. complexity += c * factor; .)
      ]
   |  "while" "(" Condition ")"
      Statement <out c>         (. complexity += c * factor; .)
   |  "break" ";"               (. complexity = 5; .)
   |  "return" [Expr] ";"
   |  "read" "(" Designator ")" ";"
   |  "print" "(" Expr ["," number] ")" ";"
   |  Block <out complexity>
   |  ";"
   ).

FormPars  =  Type ident  {"," Type ident}.
Type         =  ident ["[" "]"].
ActPars     =  "(" [ Expr {"," Expr}] ")".
Condition  =  CondTerm {"||" CondTerm}.
CondTerm =  CondFact {"&&" CondFact}.
CondFact  =  Expr RelOp Expr.
RelOp       =  "==" | "!=" | ">" | ">=" | "<" | "<=".
Expr         =  ["-"] Term {AddOp Term}.
Term         =  Factor {MulOp Factor}.
Factor      =  Designator [ActPars]
             |  number
             |  charConst
             |  "new" ident ["[" Expr "]"]
             |  "(" Expr ")".
Designator=  ident {"." ident | "[" Expr "]"}.
AddOp      =  "+" | "-".
MulOp      =  "*" | "/" | "%".

END Metrics.
```

In MethodDecl erhalten wir die Komplexität des Methodenblocks als Ausgangsattribut von Block und geben sie zusammen mit den Namen der Methode aus.

In Block addieren wir die Komplexitäten der darin enthaltenen Anweisungen und geben das Ergebnis als Ausgangsattribut zurück.

In Statement berechnen wir die Komplexität einer einzelnen Anweisung, die für die meisten Anweisungsarten 1 ist, weshalb wir complexity standardmäßig auf 1 setzen und nur in den Anweisungen ändern, die eine andere Komplexität als 1 haben. if- und while-Anweisungen haben geschachtelte Anweisungen, deren Komplexität wir durch das Ausgangsattribut c erhalten, welches wird mit dem Faktor 2 multiplizieren und zur Komplexität der if- oder while-Anweisung addieren. Eine Anweisung kann auch ein Block sein, der die Komplexität der in ihm enthaltenen Anweisungen durch ein Ausgangsattribut liefert.

Mehr ist nicht zu tun! Wir brauchen noch ein Hauptprogramm, das den Namen der Eingabedatei als Kommandozeilenparameter liest, einen Scanner und einen Parser erzeugt und die Syntaxanalyse startet:

```
class Metrics {
   public static void main (String[] arg) {
      Scanner scanner = new Scanner(arg[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      System.out.println(parser.errors.count + " errors detected");
   }
}
```

Wir verarbeiten die Compilerbeschreibung mit Coco/R:

```
java -jar Coco.jar Metrics.atg
```

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

```
javac Scanner.java Parser.java Metrics.java
```

und rufen dann den fertigen Komplexitätsanalysator wie folgt für ein MicroJava-Beispielprogramm (z.B. Eratos.mj) auf:

```
java Metrics Eratos.mj
```

Wenn Eratos.mj folgendes MicroJava-Programm enthält

```
program Eratos
   char[] sieve;
   int max;                  // maximum prime to be found
   int npp;                  // numbers per page
{
   void put (int x)
   {  if (npp == 10) {print(chr(13)); print(chr(10)); npp = 0;}
      print(x, 5);
      npp = npp + 1;
   }

   void found (int x)
      int i;
   {  put(x);
      i = x; while (i <= max) {sieve[i] = 'o'; i = i + x;}
   }

   void main()
      int i, ready;
   {  read(max); npp = 0; sieve = new char[max+1];
      i = 0; while (i <= max) {sieve[i] = 'x'; i = i + 1;}
      i = 2;
      while (i <= max) {
         found(i);
         ready = 0;
         while (ready == 0) {
            if (i > max) ready = 1;
            else if (sieve[i] == 'x') ready = 1;
            else i = i + 1;
         }
      }
   }
}
```

liefert der Aufruf von Metrics folgendes Ergebnis:

```
put: 9
found: 8
main: 71
```