**Abgabetermin: in der KW 50**

☐ **Gr. 1,** Dr. H. Dobler    **Name** _____    **Aufwand in h** _____

☐ **Gr. 2,** Dr. G. Kronberger

**Punkte** _____    **Übungsleiter** _____

Im Moodle-Kurs finden Sie unter Flex und Bison zwei ZIP-Dateien mit den GNU-Implementierungen von lex u. yacc, für Windows. Bei UNIX sind beide Werkzeuge üblicherweise bereits enthalten oder können leicht nachgeladen werden.

## 1. MiniC: Scanner und Parser mit lex und yacc    (8 Punkte)

MiniC ist eine kleine Teilmenge von C, angelehnt an MiniPascal. Unten links ist ein einfaches Programm zur Berechnung des Satzes von Pythagoras dargestellt, rechts die Grammatik von MiniC (die Sie auch im Moodle-Kurs in der Datei *MiniC.syn* finden):

```
void main() {                       MC =     "void" "main" "(" ")" "{"
  int a, b, cs;                               [ VarDecl ]
  scanf(a);                                   StatSeq
  scanf(b);                                   "}" .
  cs = (a * a) + (b * b);          VarDecl = "int" ident { "," ident } ";" .
  printf(cs);                      StatSeq = Stat { Stat } .
}                                   Stat =    [ ident "=" Expr
                                            | "scanf"  "(" ident ")"
                                            | "printf" "(" Expr  ")"
                                            ] ";" .
                                    Expr =    Term { ( "+" | "-" ) Term } .
                                    Term =    Fact { ( "*" | "/" ) Fact } .
                                    Fact =    ident | number | "(" Expr ")" .
```

Erzeugen Sie mit lex einen lexikalischen Analysator (*scanner*) und mit yacc einen Syntaxanalysator (*parser*) für MiniC und bauen Sie daraus ein Programm für die Analyse von MiniC-Programmen.

## 2. MiniCpp: Scanner und Parser mit lex und yacc    (10 + 6 Punkte)

Wir werden uns intensiver mit der etwas größeren Sprache MiniCpp beschäftigen, mit der man auch etwas anspruchsvollere Programme schreiben kann, z. B. für das Sieb des Erathostenes:

```
void Sieve(int n); // declaration       cout << 2 << " ";
                                        col = 1;
void main() {                           i = 3;
  int n;                                while (i <= n) {
  cout << "n > ";                         if (sieve[i]) {
  cin >> n;                                 if (col == 10) {
  if (n > 2)                                  cout << endl;
    Sieve(n);                                 col = 0;
} // main                                   } // if
                                          col++;
void Sieve(int n) { // definition         cout << i  << " ";
  int   col, i, j;                        j = i * i;
  bool *sieve = 0;                        while (j <= n) {
  sieve = new bool[n + 1];                  sieve[j] = false;
  i = 2;                                    j = j + 2 * i;
  while (i <= n) {                        } // while
    sieve[i] = true;                    } // if
    i++;                                i = i + 2;
  } // while                          } // while
                                      delete[] sieve;
                                    } // Sieve
```

Hier die Grammatik für MiniCpp, die Sie im Moodle-Kurs auch in der Datei *MiniCpp.syn* finden:

```
MiniCpp =      { ConstDecl | VarDef | FuncDecl | FuncDef } .
ConstDecl =    'const' Type ident Init ';' .
Init =         '=' ( false | true | number ) .
VarDef =       Type [ '*' ] ident [ Init ]
               { ',' [ '*' ] ident [ Init ] } ';' .
FuncDecl =     FuncHead ';' .
FuncDef =      FuncHead Block .
FuncHead =     Type [ '*' ] ident '(' [ FormParList ] ')' .
FormParList = ( 'void' |
                 Type [ '*' ] ident [ '[' ']' ]
                 { ',' Type [ '*' ] ident [ '[' ']' ] } ) .
Type =         'void' | 'bool' | 'int' .
Block =        '{' { ConstDecl | VarDef | Stat } '}' .
Stat =         ( IncStat | DecStat | AssignStat
               | CallStat | IfStat
               | WhileStat | BreakStat
               | InputStat | OutputStat | DeleteStat | ReturnStat
               | Block
               | ';'
               ) .
IncStat =      ident '++' ';' .
DecStat =      ident '--' ';' .
AssignStat =   ident [ '[' Expr ']' ] '=' Expr ';' .
CallStat =     ident '(' [ ActParList ] ')' ';' .
ActParList =   Expr { ',' Expr } .
IfStat =       'if' '(' Expr ')' Stat [ 'else' Stat ] .
WhileStat =    'while' '(' Expr ')' Stat .
BreakStat =    'break' ';' .
InputStat =    'cin'  '>>' ident ';' .
OutputStat =   'cout' '<<' ( Expr | string | 'endl' )
                    { '<<' ( Expr | string | 'endl' ) } ';' .
DeleteStat =   'delete' '[' ']' ident ';' .
ReturnStat =   'return' [ Expr ] ';' .
Expr =         OrExpr .
OrExpr =       AndExpr { '||' AndExpr } .
AndExpr        RelExpr { '&&' RelExpr } .
RelExpr =      SimpleExpr
               [ ( '==' | '!=' | '<' | '<=' | '>' | '>=' )
                 SimpleExpr ] .
SimpleExpr =   [ '+' | '-' ]
               Term    { ( '+' | '-' )        Term    } .
Term =         NotFact { ( '*' | '/' | '%' )  NotFact } .
NotFact =      [ '!' ] Fact .
Fact =          'false' | 'true'
               | number
               | ident [ ( '[' Expr           ']' )
                       | ( '(' [ ActParList ] ')' )
                       ]
               | 'new' Type '[' Expr ']'
               | '(' Expr ')' .
```

a)  Erzeugen Sie mit lex/flex einen lexikalischen Analysator (*scanner*) und mit yacc/bison einen Syntaxanalysator (*parser*) für MiniCpp und bauen Sie daraus ein Programm für die lexikalische und syntaktische Analyse von MiniCpp-Programmen.

b)  Erweitern Sie Ihre Grammatik aus a) zu einer ATG, sodass der (statische) Funktionsaufrufgraph des analysierten Programms erstellt wird. Gehen Sie so vor, dass von der ATG eine Textdatei (.gv) für *GraphViz* (www.graphviz.org) erzeugt wird, die mit *GVEdit* (oder mit *dot.exe* direkt oder über www.webgraphviz.com) in eine Abbildung umgesetzt werden kann.

# 1 MiniC: Scanner und Parser mit lex und yacc

Listing 1: minic.l

```
1  /* Lexical specification of the MiniC language */
2  %{
3
4    #include "minic.tab.h"
5
6    extern int yylval;
7    extern int yylineno;
8
9  %}
10
11 %%
12
13 void                 { return VOID;  }
14 main                 { return MAIN;  }
15 int                  { return INT;   }
16 scanf                { return SCANF; }
17 printf               { return PRINTF;}
18
19 [ \t\n\r]+           { ; }                    /*ignore white space: blanks and tabs      */
20 "/""/"[^\n\r]*       { ; }                    /* skip line comments */
21 [0-9]+               { return NUMBER;}
22 [a-zA-Z][_a-zA-Z0-9]* { return IDENT; }
23 .                    { return yytext[0]; }   /*return all other chars as tokens to parser: '+',
   ↪   ...       */
24
25 %%
26
27 int yywrap() {
28   return 1;
29 }
```

Listing 2: minic.y

```
1  /* The grammar specification of the MiniC language */
2  %{
3    #include <stdio.h>
4    #include <stdlib.h>
5
6    extern int yylineno;
7    extern int yylval;
8    extern FILE *yyin;
9
10   int yylex(void);
11   int yyerror(char *msg);
12 %}
13
14 %token INT
15 %token MAIN
16 %token PRINTF
17 %token SCANF
18 %token VOID
19 %token IDENT
20 %token NUMBER
21
22 %%
23 MC:
24   VOID MAIN '(' ')' '{' OptVarDecl StateSeq '}'
```

```
25      ;
26  OptVarDecl:
27    /* EPSILON */
28    | VarDecl
29    ;
30  VarDecl:
31    INT IdentList ';'
32    ;
33  IdentList:
34    IDENT
35    | IdentList ',' IDENT
36    ;
37  StateSeq:
38    Stat StatList
39    ;
40  StatList:
41    /* EPSILON */
42    | StatList Stat
43    ;
44  Stat:
45    OptStat ';'
46    ;
47  OptStat:
48    /* EPSILON */
49    | IDENT '=' Expr
50    | SCANF '(' IDENT ')'
51    | PRINTF'(' Expr ')'
52    ;
53  Expr:
54    Term TermList
55    ;
56  TermList:
57    /* EPSILON */
58    | TermList Sign Term
59    ;
60  Sign:
61    '+'
62    | '-'
63    ;
64  Term:
65    Fact FactList
66    ;
67  FactList:
68    /* EPSILON */
69    | FactList Operator Fact
70    ;
71  Operator:
72    '*'
73    | '/'
74    ;
75  Fact:
76    IDENT
77    | NUMBER
78    | '(' Expr ')'
79    ;
80  %%
81
82  int yyerror(char *msg){
83      printf("syntac error in line %i: %s\n", yylineno, msg);
84      return 0;
85  }
86
87  int main (int argc, char** argv) {
```

```
88   //if(argc > 1) {
89     FILE *fin=fopen("../test.c", "r");// fopen(argv[1], "r");
90     if(fin != NULL){
91       yyin = fin;
92     } else {
93       printf("Error druing file read");
94       exit(EXIT_FAILURE);
95     }
96   //}
97   yydebug = 1;
98   if(yyparse() == 0){
99       printf("Successfully parsed file");
100     }else{
101       printf("Parsing of file failed");
102     }
103   printf("\n\n%d lines analysed\n", yylineno);
104
105   return 0;
106 }
```

## 1.1 MiniC Tests

Listing 3: Testprogramm für Parser

```
1 void main() {
2     int a, b, cs;
3     scanf(a);
4     scanf(b);
5     cs = (a * a) + (b * b);
6     printf(cs);
7 }
```

```
Entering state 28
Reducing stack by rule 25 (line 80):
    $1 = token IDENT ()
-> $$ = nterm Fact ()
Stack now 0 1 3 5 6 7 9 17 25 15 23 32 39 45 30 33 40 48
Entering state 50Successfully parsed file


Reducing stack by rule 22 (line 73):
1 lines analysed
```

Abbildung 1: Ausgabe Testprogramms für Parser

## 2 MiniCpp: Scanner und Parser mit lex und yacc

Listing 4: minicpp.l

```
1   /* MiniCpp lexical scanner specification */
2   %{
3       #include "minicpp.tab.h"
4   %}
5
6   %%
7
8   [ \t]+  { ; }  /* ignore whitespace and tabs */
9   "/*"    {
10              /* skip multiline comments */
11              int prevChar = 0;
12              int ch = input();
13              for(;;){
14                if(ch == EOF) { /* common unclosed and end of file found */
15                  break;
16                } /* if */
17                if(ch == '\n') { /* Multiline comment, new line found */
18                  yylineno++;
19                } else if ((prevChar == '*') && (ch == '/')) { /* correct comment ending */
20                  break;
21                } /* if */
22                prevChar = ch;
23                ch = input();
24              } /* for */
25            }
26   "//"    {
27              /* skip single line comments */
28              int ch = input();
29              for(;;){
30                if(ch == EOF) { /* comment contains end of file */
31                  break;
32                } /* if */
33                if(ch == '\n') { /* single line comment new line found */
34                  yylineno;
35                  break;
36                } /* if */
37                ch = input();
38              } /* for */
39            }
40
41
42   void   { return VOID;   }
43   int    { return INT;    }
44   bool   { return BOOL;   }
45   string { return STRING; }
46   cout   { return COUT;   }
47   cin    { return CIN;    }
48   endl   { return ENDL;   }
49   ident  { return IDENT;  }
50   number { return NUMBER; }
51   true   { return TRUE;   }
52   false  { return FALSE;  }
53   const  { return CONST;  }
54   if     { return IF;     }
55   else   { return ELSE;   }
56   while  { return WHILE;  }
57   break  { return BREAK;  }
58   return { return RETURN; }
59   new    { return NEW;    }
```

```
60  delete { return DELETE; }
61
62  "||"  { return OR;      }
63  "&&"  { return AND;     }
64  ">>"  { return SHIFTR; }
65  "<<"  { return SHIFTL; }
66  "="   { return ASSIGN; }
67  "!="  { return NEQ;    }
68  "=="  { return EQ;      }
69  "<="  { return LE;      }
70  ">="  { return GE;      }
71  "<"   { return LT;      }
72  ">"   { return GT;      }
73  "++"  { return INC;    }
74  "--"  { return DEC;    }
75  "+"   { return PLUS;   }
76  "-"   { return MINUS;  }
77  "*"   { return ASTERIX;}
78  "/"   { return DIV;    }
79  "%"   { return MOD;    }
80  "!"   { return SIGNCALL;}
81
82  [a-zA-Z][a-zA-Z0-9]*    { yylval = strdup(yytext); return IDENT; }
83  \"[^\"]*\"              { ; return STRING;  }
84  [0-9]+                  { ; return NUMBER;  }
85  \n                      { yylineno++;       }
86  .                       { return yytext[0]; } /* all other chars as tokens for parsers */
87
88  %%
89
90  int yywrap() {
91    return 1;   /* on end no further files to scan */
92  } /*int  yywrap */
```

Listing 5: minicpp.y

```
1   /* Parser specification */
2   %{
3     #ifndef YYSTYPE
4     #define YYSTYPE char*
5     #endif
6     #include <stdio.h>
7     #include <stdlib.h>
8     #include <string.h>
9
10    // Extern members
11    extern int yylineno;
12    extern FILE *yyin;
13    extern char* yylval;
14
15    // Variable declarations
16    FILE *outFile = NULL; // The output file
17    char *methodArr[255]; // Array of calls per method. TODO Should be dynamic.
18    int curLen  = 0;          // Current methodArr index
19
20    int yylex(void);
21    int yyerror(char *msg);
22  %}
23
24  %token VOID
25  %token INT
26  %token BOOL
```

```
27  %token STRING
28  %token COUT
29  %token CIN
30  %token ENDL
31  %token IDENT
32  %token NUMBER
33  %token TRUE
34  %token FALSE
35  %token CONST
36  %token IF
37  %token ELSE
38  %token WHILE
39  %token BREAK
40  %token RETURN
41  %token NEW
42  %token DELETE
43  %token OR
44  %token AND
45  %token SHIFTR
46  %token SHIFTL
47  %token ASSIGN
48  %token NEQ
49  %token EQ
50  %token LE
51  %token GE
52  %token INC
53  %token DEC
54  %token ADD
55  %token SUBST
56  %token LT
57  %token GT
58  %token PLUS
59  %token MINUS
60  %token ASTERIX
61  %token DIV
62  %token MOD
63  %token SIGNCALL
64
65  %%
66  MiniCpp:
67    /* EPSILON */
68    | MiniCpp ConstDecl
69    | MiniCpp VarDef
70    | MiniCpp FuncDecl
71    | MiniCpp FuncDef
72    ;
73  ConstDecl:
74      CONST Type IDENT Init ';'
75      ;
76  Init:
77      ASSIGN TypeVal
78      ;
79  TypeVal:
80    FALSE
81    | TRUE
82    | NUMBER
83    ;
84  VarDef:
85    Type OptAsterixList IDENT OptInit VarDefList ';'
86    ;
87  VarDefList:
88    /* EPSILON */
89    | VarDefList ',' OptAsterixList IDENT OptInit
```

```
 90       ;
 91   OptAsterixList:
 92     /* EPSILON */
 93     | OptAsterixList ASTERIX
 94     ;
 95   OptInit:
 96     /* EPSILON */
 97     | Init
 98     ;
 99   FuncDecl:
100     FuncHead ';'
101     ;
102   FuncDef:
103     FuncHead Block  {
104                     // iterate over all current remembered methods
105                     for(int i=0;i<curLen;i++){
106                       // Write to dot system
107                       fprintf(outFile,"%s -> %s\n",$1,methodArr[i]);
108                       // Free string produced by strdup
109                       free(methodArr[i]);
110                     }
111                     // Reset current methodArr index
112                     curLen=0;
113                   }
114     ;
115   FuncHead:
116     Type OptAsterixList IDENT '(' FormParList ')' {$$ = $4;}
117     ;
118   FormParList:
119     /* EPSILON */
120     | VOID
121     | Type OptAsterixList IDENT OptBrackets FormParListList
122     ;
123   OptBrackets:
124     /* EPSILON */
125     | '[' ']'
126     ;
127   FormParListList:
128     /* EPSILON */
129     | FormParListList ',' Type OptAsterixList IDENT OptBrackets
130     ;
131   Type:
132     VOID
133     | BOOL
134     | INT
135     ;
136   Block:
137     '{' BlockList '}'
138     ;
139   BlockList:
140     /* EPSILON */
141     | BlockList ConstDecl
142     | BlockList VarDef
143     | BlockList Stat
144     ;
145   Stat:
146     IncStat
147     | DecStat
148     | AssignStat
149     | CallStat
150     | IfStat
151     | WhileStat
152     | BreakStat
```

```
153 |     | InputStat
154 |     | OutputStat
155 |     | DeleteStat
156 |     | ReturnStat
157 |     | Block
158 |     | ';'
159 |     ;
160 | IncStat:
161 |     IDENT INC ';'
162 |     ;
163 | DecStat:
164 |     IDENT DEC ';'
165 |     ;
166 | AssignStat:
167 |     IDENT AssignOptExpr ASSIGN Expr ';'
168 |     ;
169 | AssignOptExpr:
170 |     /* EPSILON */
171 |     | '[' Expr ']'
172 |     ;
173 | CallStat:
174 |     IDENT '(' OptActParList ')' ';' { methodArr[curLen]=$1; curLen++; }
175 |     ;
176 | OptActParList:
177 |     /* EPSILON */
178 |     | ActParList
179 |     ;
180 | ActParList:
181 |     Expr ExprList
182 |     ;
183 | ExprList:
184 |     /* EPSILON */
185 |     | ExprList ',' Expr
186 |     ;
187 | IfStat:
188 |     IF '(' Expr ')' Stat ElseStatOpt
189 |     ;
190 | ElseStatOpt:
191 |     /* EPSILON */
192 |     | ELSE Stat
193 |     ;
194 | WhileStat:
195 |     WHILE '(' Expr ')' Stat
196 |     ;
197 | BreakStat:
198 |     BREAK ';'
199 |     ;
200 | InputStat:
201 |     CIN SHIFTR IDENT ';'
202 |     ;
203 | OutputStat:
204 |     COUT SHIFTL OptOutputStat OutputStatList ';'
205 |     ;
206 | OptOutputStat:
207 |     Expr
208 |     | STRING
209 |     | ENDL
210 |     ;
211 | OutputStatList:
212 |     /* EPSILON */
213 |     | OutputStatList SHIFTL OptOutputStat
214 |     ;
215 | DeleteStat:
```

```
216        DELETE '[' ']' IDENT ';'
217      ;
218  ReturnStat:
219      RETURN OptExpr ';'
220      ;
221  OptExpr:
222      /* EPSILON */
223    | Expr
224      ;
225  Expr:
226      OrExpr
227      ;
228  OrExpr:
229      AndExpr OrExprList
230      ;
231  OrExprList:
232      /* EPSILON */
233    | OrExprList OR AndExpr
234      ;
235  AndExpr:
236      RelExpr AndExprList
237      ;
238  AndExprList:
239      /* EPSILON */
240    | AndExprList AND RelExpr
241      ;
242  RelExpr:
243      SimpleExpr OptRelExpr
244      ;
245  OptRelExpr:
246      /* EPSILON */
247    | OptOperator SimpleExpr
248      ;
249  OptOperator:
250      EQ
251    | NEQ
252    | LT
253    | LE
254    | GT
255    | GE
256      ;
257  SimpleExpr:
258      OptSign Term SimpleExprList
259      ;
260  Sign:
261      PLUS
262    | MINUS
263      ;
264  OptSign:
265      /* EPSILON */
266    | Sign
267      ;
268  SimpleExprList:
269      /* EPSILON */
270    | SimpleExprList Sign Term
271      ;
272  Term:
273      NotFact TermList
274      ;
275  TermList:
276      /* EPSILON */
277    | TermList OptNotFactOperator NotFact
278      ;
```

```
279  OptNotFactOperator:
280      ASTERIX
281      | DIV
282      | MOD
283      ;
284  NotFact:
285      OptCallSign Fact
286      ;
287  OptCallSign:
288      /* EPSILON */
289      | SIGNCALL
290      ;
291  Fact:
292      FALSE
293      | TRUE
294      | NUMBER
295      | IDENT OptIdentFact
296      | NEW Type '[' Expr ']'
297      | '(' Expr ')'
298      ;
299  OptIdentFact:
300      /* EPSILON */
301      | '[' Expr ']'
302      | '(' OptActParList ')'
303      ;
304
305  %%
306
307  int yyerror(char *msg) {
308      printf("syntac error in line %i: %s\n", yylineno, msg);
309      return 0;
310  } /*yyerror*/
311
312
313  int main(int argc, char *argv[]) {
314      //if(argc > 1) {
315          FILE *f_handle = fopen("../test.cpp","r"); // fopen(argv[1],"r");
316          if(f_handle != NULL){
317              yyin = f_handle;
318          } else {
319              printf("ERROR: file not found");
320              return EXIT_FAILURE;
321          }
322      //}
323
324      //grahviz output file
325      outFile = fopen("graph.gv","w");
326      if(outFile == NULL){
327          printf("ERROR: could not open output file graph.gv");
328              return EXIT_FAILURE;
329      }
330
331      // write graphviz init line to file
332      fprintf(outFile, "digraph G {");
333
334      yydebug = 1;
335      if(yyparse() == 0) {
336          // write graphviz closing bracket to file
337          fprintf(outFile, "}");
338          fclose(outFile);
339
340          printf("successfully parsed %i lines\n", yylineno);
341          return 0;
```

```
342        }
343
344        // write graphviz closing bracket to file
345        fprintf(outFile, "}");
346        fclose(outFile);
347
348        return 1;
349    } /*main*/
```

Listing 6: Testprogramm für Parser

```
1   /**
2    * Implements the algorithm sieve of Eratosthenes.
3    * @param n the count of items to use
4    */
5   void Sieve(int n);
6
7   /**
8    * Main entry of the program.
9    */
10  void main() {
11      int n;
12      cout << "n > ";
13      cin >> n;
14      if (n > 2) {
15          Sieve(n);
16      } // if
17  } // main
18
19  // Implementation of sieve
20  void Sieve(int n) {
21      int col, i, j;
22      bool *sieve = 0;
23      sieve = new bool[n + 1];
24      i = 2;
25      while (i <= n) {
26          sieve[i] = true;
27          i++;
28      } // while
29      cout << 2 << " ";
30      col = 1;
31      i = 3;
32      while (i <= n) {
33          if (sieve[i]) {
34              if (col == 10) {
35                  cout << endl;
36                  col = 0;
37              } // if
38              col++;
39              cout << i << " ";
40              j = i * i;
41              while (j <= n) {
42                  sieve[j] = false;
43                  j = j + 2 * i;
44              } // while
45          } // if
46          i = i + 2;
47      } // while
48      delete[] sieve;
49  } // Sieve
```

Listing 7: Testprogramm für Graphviz Test

```
1   // Function declaration
2   void m1();
3
4   void m2();
5
6   void m3();
7
8   // function definition
9   void m1() {
10      m2();
11  }
12
13  void m2() {
14      m3();
15  }
16
17  void m3() {
18      m4();
19  }
20
21  void m4(int n) {
22      // do nothing
23      if (n < 10) {
24          m4(n + 1);
25      }
26  }
27
28  void main() {
29      m1();
30      m2();
31      m3();
32      m4();
33  }
```

## 2.1 MiniCPP Tests

```
Stack now 0 1 2
Cleanup: popping token $end ()
Cleanup: popping nterm MiniCpp ()
successfully parsed 41 lines

Process finished with exit code 0
```

Abbildung 2: Ausgabe Testprogramm für den Parser

```
Entering state 91
Next token is token ')' ()
Shifting token ')' ()
Entering state 121
Reading a token: Next token is token ';' ()
successfully parsed 31 lines
Shifting token ';' ()
Entering state 144
```

Abbildung 3: Ausgabe Testprogramm für Graphviz

Listing 8: Generierte Graphviz Datei

```
1  digraph G {m1 -> m2
2  m2 -> m3
3  m3 -> m4
4  m4 -> m4
5  main -> m1
6  main -> m2
7  main -> m3
8  main -> m4
9  }
```
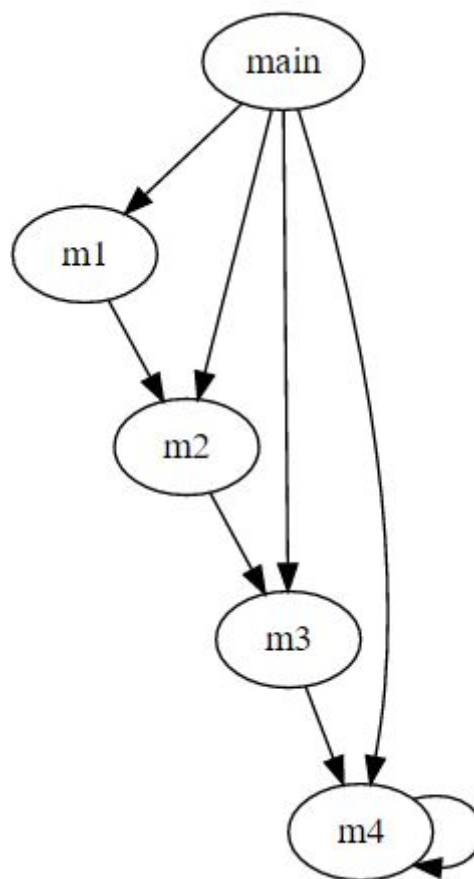


Abbildung 4: Statischer Aufrufgraph des zweiten Testprogramms