FCW 1x Übung zu Formale Sprachen, Compiler- und Werkzeugbau 1 — WS 2016/17, Übung 4

Abgabetermin: in der KW 50

☐ **Gr. 1,** Dr. H. Dobler
☐ **Gr. 2,** Dr. G. Kronberger

Name _____ **Aufwand in h** _____

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() {
  int a, b, cs;
  scanf(a);
  scanf(b);
  cs = (a * a) + (b * b);
  printf(cs);
}
```

```
MC =      "void" "main" "(" ")" "{"
          [ VarDecl ]
          StatSeq
          "}" .
VarDecl = "int" ident { "," ident } ";" .
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

void main() {
  int n;
  cout << "n > ";
  cin >> n;
  if (n > 2)
    Sieve(n);
} // main

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

```
/* Lexical specification of the MiniC language */
%{

  #include "minic.tab.h"

  extern int yylval;
  extern int yylineno;

%}

%%

void                    { return VOID;  }
main                    { return MAIN;  }
int                     { return INT;   }
scanf                   { return SCANF; }
printf                  { return PRINTF;}

[ \t\n\r]+               { ; }                  /*ignore white space: blanks and tabs       */
"/""/"[^\n\r]*          { ; }                  /* skip line comments */
[0-9]+                  { return NUMBER;}
[a-zA-Z][_a-zA-Z0-9]*   { return IDENT; }
.                       { return yytext[0]; }  /*return all other chars as tokens to parser: '+',
 ↪  ...      */

%%

int yywrap() {
  return 1;
}
```

Listing 2: minic.y

```
/* The grammar specification of the MiniC language */
%{
  #include <stdio.h>
  #include <stdlib.h>

  extern int yylineno;
  extern int yylval;
  extern FILE *yyin;

  int yylex(void);
  int yyerror(char *msg);
%}

%token INT
%token MAIN
%token PRINTF
%token SCANF
%token VOID
%token IDENT
%token NUMBER

%%
MC:
  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

```
/* MiniCpp lexical scanner specification */
%{
  #include "minicpp.tab.h"
%}

%%

[ \t]+  { ; }  /* ignore whitespace and tabs */
"/*"    {
          /* skip multiline comments */
          int prevChar = 0;
          int ch = input();
          for(;;){
            if(ch == EOF) { /* common unclosed and end of file found */
              break;
            } /* if */
            if(ch == '\n') { /* Multiline comment, new line found */
              yylineno++;
            } else if ((prevChar == '*') && (ch == '/')) { /* correct comment ending */
              break;
            } /* if */
            prevChar = ch;
            ch = input();
          } /* for */
        }
"//"    {
          /* skip single line comments */
          int ch = input();
          for(;;){
            if(ch == EOF) { /* comment contains end of file */
              break;
            } /* if */
            if(ch == '\n') { /* single line comment new line found */
              yylineno;
              break;
            } /* if */
            ch = input();
          } /* for */
        }


void   { return VOID;   }
int    { return INT;    }
bool   { return BOOL;   }
string { return STRING; }
cout   { return COUT;   }
cin    { return CIN;    }
endl   { return ENDL;   }
ident  { return IDENT;  }
number { return NUMBER; }
true   { return TRUE;   }
false  { return FALSE;  }
const  { return CONST;  }
if     { return IF;     }
else   { return ELSE;   }
while  { return WHILE;  }
break  { return BREAK;  }
return { return RETURN; }
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
```

```
%token STRING
%token COUT
%token CIN
%token ENDL
%token IDENT
%token NUMBER
%token TRUE
%token FALSE
%token CONST
%token IF
%token ELSE
%token WHILE
%token BREAK
%token RETURN
%token NEW
%token DELETE
%token OR
%token AND
%token SHIFTR
%token SHIFTL
%token ASSIGN
%token NEQ
%token EQ
%token LE
%token GE
%token INC
%token DEC
%token ADD
%token SUBST
%token LT
%token GT
%token PLUS
%token MINUS
%token ASTERIX
%token DIV
%token MOD
%token SIGNCALL

%%
MiniCpp:
  /* EPSILON */
  | MiniCpp ConstDecl
  | MiniCpp VarDef
  | MiniCpp FuncDecl
  | MiniCpp FuncDef
  ;
ConstDecl:
    CONST Type IDENT Init ';'
    ;
Init:
    ASSIGN TypeVal
    ;
TypeVal:
  FALSE
  | TRUE
  | NUMBER
  ;
VarDef:
  Type OptAsterixList IDENT OptInit VarDefList ';'
  ;
VarDefList:
  /* EPSILON */
  | 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   ;
```

```
OptNotFactOperator:
  ASTERIX
  | DIV
  | MOD
  ;
NotFact:
  OptCallSign Fact
  ;
OptCallSign:
  /* EPSILON */
  | SIGNCALL
  ;
Fact:
  FALSE
  | TRUE
  | NUMBER
  | IDENT OptIdentFact
  | NEW Type '[' Expr ']'
  | '(' Expr ')'
  ;
OptIdentFact:
  /* EPSILON */
  | '[' Expr ']'
  | '(' OptActParList ')'
  ;

%%

int yyerror(char *msg) {
    printf("syntac error in line %i: %s\n", yylineno, msg);
    return 0;
} /*yyerror*/


int main(int argc, char *argv[]) {
    //if(argc > 1) {
        FILE *f_handle = fopen("../test.cpp","r"); // fopen(argv[1],"r");
        if(f_handle != NULL){
            yyin = f_handle;
        } else {
            printf("ERROR: file not found");
            return EXIT_FAILURE;
        }
    //}

    //grahviz output file
    outFile = fopen("graph.gv","w");
    if(outFile == NULL){
        printf("ERROR: could not open output file graph.gv");
            return EXIT_FAILURE;
    }

    // write graphviz init line to file
    fprintf(outFile, "digraph G {");

    yydebug = 1;
    if(yyparse() == 0) {
        // write graphviz closing bracket to file
        fprintf(outFile, "}");
        fclose(outFile);

        printf("successfully parsed %i lines\n", yylineno);
        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

```
// Function declaration
void m1();

void m2();

void m3();

// function definition
void m1() {
    m2();
}

void m2() {
    m3();
}

void m3() {
    m4();
}

void m4(int n) {
    // do nothing
    if (n < 10) {
        m4(n + 1);
    }
}

void main() {
    m1();
    m2();
    m3();
    m4();
}
```

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

```
digraph G {m1 -> m2
m2 -> m3
m3 -> m4
m4 -> m4
main -> m1
main -> m2
main -> m3
main -> m4
}
```

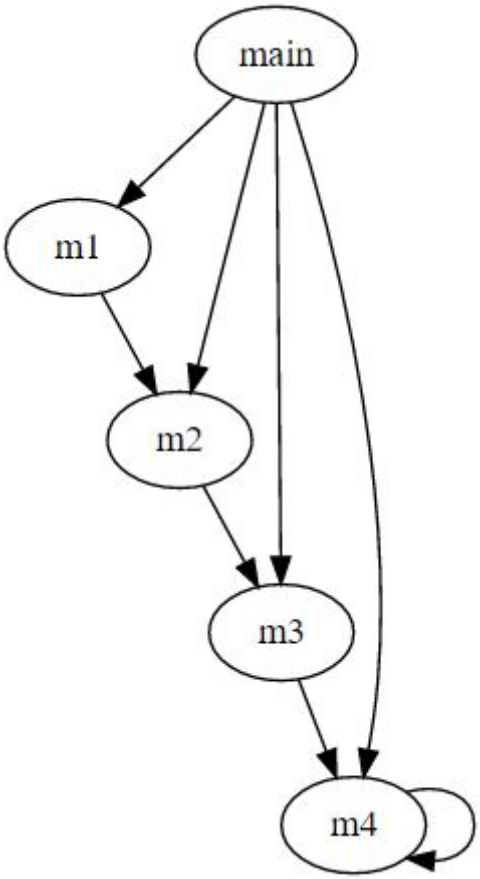

Abbildung 4: Statischer Aufrufgraph des zweiten Testprogramms