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.