FCW 1x Übung zu Formale Sprachen, Compiler- und Werkzeugbau 1 — WS 2022/23, Übung 4

Abgabetermin: in der KW 51

☐ **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/Linux sind beide üblicherweise bereits enthalten oder können leicht nachgeladen werden.

1. MiniC: Scanner und Parser mit (f)lex und yacc/bison (6 Punkte)

MiniC ist eine kleine Teilmenge von C, angelehnt an MiniPascal. Unten links ist ein einfaches MiniC-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/flex einen lexikalischen Analysator (*scanner*), mit yacc/bison einen Syntaxanalysator (*parser*) für MiniC und bauen Sie daraus ein Analyseprogramm für MiniC-Programme.

2. MiniCpp: Scanner und Parser mit (f)lex und yacc/bison UND … (12 + 6 Punkte)

Wir werden uns aber intensiver mit der etwas größeren Sprache MiniCpp beschäftigen, mit der man auch 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 = nullptr;
  sieve = new bool[n + 1];
  i = 2;
  while (i <= n) {
    sieve[i] = true;
    i++;
  } // while
  // continued on the right side
  cout << 2 << "\t";
  col = 1;
  i = 3;
  while (i <= n) {
    if (sieve[i]) {
      if (col == 10) {
        cout << endl; // same as "\n"
        col = 0;
      } // if
      ++col;
      cout << i << "\t";
      j = i * i;
      while (j <= n) {
        sieve[j] = false;
        j += 2 * i;
      } // while
    } // if
    i += 2;
  } // while
  delete[] sieve;
} // Sieve
```

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

```
MiniCpp =      { ConstDef | VarDef | FuncDecl | FuncDef | ';' } .
ConstDef =     'const' Type ident Init { ',' ident Init } ';' .
Init =         '=' ( false | true | 'nullptr'
                   | [ '+' | '-' ] 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 =        '{' { ConstDef | VarDef | Stat } '}' .
Stat =         ( EmptyStat  | BlockStat  | ExprStat
               | IfStat     | WhileStat  | BreakStat
               | InputStat  | OutputStat
               | DeleteStat | ReturnStat
               ) .
EmptyStat =    ';' .
BlockStat =    Block .
ExprStat =     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 } .
OrExpr =       AndExpr { '||' AndExpr } .
AndExpr        RelExpr { '&&' RelExpr } .
RelExpr =      SimpleExpr
               { ( '==' | '!=' | '<' | '<=' | '>' | '>=' ) SimpleExpr } .
SimpleExpr =   [ '+' | '-' ]
               Term    { ( '+' | '-' )       Term    } .
Term =         NotFact { ( '*' | '/' | '%' ) NotFact } .
NotFact =      [ '!' ] Fact .
Fact =         ( 'false' | 'true' | 'nullptr' | number
               | [ '++' | '--' ]
                 ident [ ( '[' Expr           ']' )
                       | ( '(' [ ActParList ] ')' )
                       ]
                 [ '++' | '--' ]
               | 'new' Type '[' Expr ']'
               |  '(' Expr ')'
               ) .
ActParList =   Expr { ',' Expr } .
```

a) Erzeugen Sie mit lex/flex einen lexikalischen Analysator (*scanner*), mit yacc/bison einen Syntaxanalysator (*parser*) für MiniCpp, und bauen Sie daraus ein Analyseprogramm für MiniCpp-Programme.

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