FCW 1x Übung zu Formale Sprachen, Compiler- und Werkzeugbau 1 — WS 2022/23, Übung 5

Abgabetermin: in der KW 3

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

Name Stefan Weißensteiner **Aufwand in h** 4

Punkte _____ **Übungsleiter** _____

Im Moodle-Kurs finden Sie in der Datei *Coco-2.zip* die .NET-Version von Coco-2 (in und für C#). Für Coco-2 gibt es keine detaillierte Dokumentation, aber die Grundzüge sind in zwei Artikeln beschreiben, die Sie auch im Moodle-Kurs finden (im Abschnitt Sonstiges Material: *Coco-2* und *Top-Down Parsing in Coco-2*).

1. MiniC: Scanner und Parser mit Coco-2 (8 Punkte)

MiniC kennen Sie ja schon. Zur Wiederholung: 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 Coco-2 einen lexikalischen Analysator (*scanner*) und einen Syntaxanalysator (*parser*) für MiniC und bauen Sie daraus ein Programm für die Analyse von MiniC-Programmen.

2. MiniCpp: Scanner, Parser und ... mit Coco-2 (6 + 4 + 6 Punkte)

MiniCpp kennen Sie zwar auch schon, hier aber trotzdem das Beispiel zur Wiederholung:

```
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
```

Und hier noch einmal die Grammatik für MiniCpp, die Sie auch im Moodle-Kurs 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 Coco-2 einen lexikalischen Analysator (*scanner*) und einen Syntaxanalysator (*parser*) für MiniCpp.

b) Bauen Sie auf Basis von a) ein Programm für die Analyse von MiniCpp-Quelltextgen. Von diesem Werkzeug für die *statische Programmanalyse* sollen mindestens folgende Daten ermittelt werden:
   - die Anzahl der Zeilen (*lines of code*, LOC),
   - die Anzahl der Anweisungen (*statements*) und
   - die Strukturkomplexität *V* nach Thomas J. **McCabe** (*V* = 1 + Anzahl der binären Verzweigungen) berechnet werden.

   Natürlich wäre auch *EV* (die essenzielle Strukturkomplexität = *V* nach Reduktion der D-Diagrammanteile) von Interesse, vielleicht schaffen Sie das ja auch noch.

c) M. H. **Halstead** hat im Vergleich zu *V* und *EV* detailliertere Metriken vorgeschlagen, die nicht nur die Größe eines Programms (in Form seiner "Länge" *N*, s. u.) und seine Struktur (wie McCabe) in Betracht ziehen, sondern vor allem den Umfang und die Komplexität der darin durchgeführten Berechnungen berücksichtigen. Dafür müssen die vier in Tab. 0.1 definierten Werte *n1*, *N1*, *n2* und *N2* aus dem Quelltext eines Programms ermittelt werden.

	Anzahl unterschiedlicher ...	Gesamtanzahl der verwendeten ...
... Operatoren	n1	N1
... Operanden	n2	N2

Tab. 0.1: Werte zur Berechnung der Halstead-Metriken

Dabei gelten als Operatoren nicht nur die mathematischen Operatoren (z. B. die arithmetischen +, -, *, / und die relationalen ==, <, >, ...), sondern auch Symbole mit Effekt (z. B. =, nicht aber ;) sowie jene Schlüsselwörter (z. B. *if*, *else* und *while*), die Einfluss auf die Wirkungsweise eines Programms haben. Als Operanden werden alle Elemente eines Programms gewertet, die Daten repräsentieren (vor allem Literale, Konstanten, Variablen, aber auch Sprungmarken). Aus den vier Größen in Tab. 0.1 können die in Tab. 0.2 genannten fünf Halstead-Metriken berechnet werden.

Halstead-Metrik	Bezeichnung	Formel
Alphabet (engl. *vocabulary*)	n	n = n1 + n2
Länge (engl. *length*)	N	N = N1 + N2
Volumen (engl. *volume*)	V	$V = N \cdot \log_2(n)$
Schwierigkeit (engl. *difficulty*)	D	D = (n1·N2) / (2·n2)
Aufwand (engl. *effort*)	E	E = D·V

Tab. 0.2: Halstead-Metriken und ihre Berechnung

Erweitern Sie Ihr Werkzeug zur statischen Analyse von MiniCpp-Programmen aus b) um die Berechnung der oben erläuterten Halstead-Metriken.

1. MiniC: Scanner und Parser mit Coco-2

a)

Code:

```
COMPILER MiniC

CHARACTER SETS
  digit      = '0' .. '9'.
  whiteSpace = CHR(9) + EOL IGNORE. /*' ' ignored by default*/
  letter     = 'a' .. 'z' + 'A' .. 'Z' + '_'.

COMMENTS
  FROM '/*' TO '*/' .
  FROM '//' TO EOL .

KEYWORDS
  'void'. 'main'. 'int'. 'scanf'. 'printf'. 'if'. 'else'.

TOKENS
  '+'.  '-'.  '*'.  '/'.  '('.  ')'.
  '{'. '}'.
  ','. ';'. '='.

TOKEN CLASSES
  number<<out int n>> =
    digit { digit }           LEX<<n = Convert.ToInt32(tokenStr);>>.

  ident<<out string idStr>> =
    letter { letter | digit } LEX<<idStr = tokenStr;>>
  .

NONTERMINALS
  MiniC.
  VarDecl.
  StatSeq.
  Stat.
  Expr.
  Term.
  Fact.

RULES

  MiniC = 'void' 'main' '(' ')' '{'
    [ VarDecl ]
    StatSeq
    '}' .

  VarDecl =
```

```
    'int' ident<<out string id1>> { ',' ident<<out string id2>> } ';' .

  StatSeq =
    Stat { Stat } .

  Stat =
    (';'
    | ident<<out string id1>> '=' Expr ';'
    | 'scanf' '(' ident<<out string id2>> ')' ';'
    | 'printf' '(' Expr ')' ';'
    ).

  Expr =
    Term
    { '+' Term
    | '-' Term
    }.

  Term =
    Fact
    { '*' Fact
    | '/' Fact
    }.

  Fact =     LOCAL<<int f = 0; string id = "";>>
      ident<<out id>>
    | number<<out f>>
    | '(' Expr ')'.

END MiniC.
```

Commands

```
SG MiniC.atg
PGT MiniC.atg
csc *.cs
Main.exe -> SVP.mc
```

Console Output (to avoid background of command prompt)

```
C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniC>SG MiniC.atg
-----------------------------------
 C o c o - 2  SG    Version C#, 1.3
 Copyright H. Dobler     April 2020
-----------------------------------


source file in "MiniC.atg"
```

```
parsing ...
evaluating lexical structure ...
generating scanner ...
listing ...

compilation completed in 0:0.24

no errors detected

C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniC>PGT MiniC.atg
-------------------------------------
 C o c o - 2  PGT  Version  C#, 1.3
 Copyright H. Dobler     April 2020
-------------------------------------


source file in "MiniC.atg"
parsing ...
generating semantic evaluator ...
evaluating syntax ...
checking grammar ...
generating syntax analyzer ...
listing ...

compilation completed in 0:00.28
no errors detected

C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniC>csc *.cs
Microsoft (R) Visual C# Compiler version 4.4.0-6.22559.4 (d7e8a398)
Copyright (C) Microsoft Corporation. All rights reserved.


C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniC>Main.exe
---------------------------
 MiniC Compiler  Version X
 Frontend gen. with Coco-2
---------------------------


source file > SVP.mc
parsing ...
no errors detected

[c]ontinue or [q]uit >
```

2. MiniCpp: Scanner, Parser und ... mit Coco-2

Lösungsidee a)

Die Grammatik aus der Angabe 1:1 nach dem Schlüsselwort **RULES** reinkopieren, alle Regeln in **NONTERMINALS** reinschreiben, für **string** eine **TOKEN CLASS** definieren und **KEYWORDS** und **TOKENS** ergänzen. Für die Terminal-Klasse **string** habe ich im Paper über Coco-2 gelesen, wie man ein **CHARACTER SET** für alle Zeichen außer einem Bestimmten definieren kann (alle Zeichen außer ein darauffolgendes **"** mit **anyButDoubleQuote = ANY - '"'.**)

Beim erstmaligen Generieren des Parsers kamen jedoch einige LL(1) Fehler auf. Nach durchgehen der Grammatik bin auch folgende Ursachen gestoßen:

- bei der Regel **FormParList** kann ein Typ entweder nur **'void'** sein, oder aus mehreren **Type ['*'] ident ['[' ']']** bestehen. Da aber **Type** auch **'void'** beinhält und beide in jeweils einer Alternative stehen, reicht 1 Vorgriffssymbol nicht aus.
- bei der Regel **MiniCpp** fangen die Alternativen **VarDef**, **FuncDef** und **FuncDecl** mit den Terminal-Symbolen **Type ['*'] ident** an. Hier würden nicht einmal 3 Vorgriffssymbole ausreichen.
- der übliche Verdächtige: dangling else

Die Grammatik wurde dann in eine äquivalente Grammatik transformiert, die alle LL(1)-Konflikte außer dem dangling else (dazu müssten wir ja die Sprache verändern) eliminiert.

Lösungsidee b)

- LOCs bekommen wir von **MiniCppLex.curLine** in **MiniCppSyn.cs**
- für die Anzahl der Statements muss man nur bei jedem erkannten **Stat** einen Zähler inkrementieren
- Srtrukturkomplexität: die Anzahl der Schleifen/Verzweigunen (if/while) zählen + 1.
- Die Ermittlung der essentiellen Strukturkomplexität habe ich nicht umgesetzt, da sich das für mich aus der Angabe wie eine optionale Zusatzaufgabe angehört hat. Falls von uns erwartet wurde, dies auch zu tun, dann bitte eindeutiger formulieren.

Lösungsidee c)

Operatoren mit Effekt:

- alle Operatoren in Regeln **Expr, RelExpr, Notfact, Term**
- '+', '-' in Regel **SimpleExpr**
- '!', '||', '&&', '++', '--'
- Schlüsselwörter 'if', 'while', 'else', 'break', 'cin', 'cout'
- laut https://en.wikipedia.org/wiki/Halstead_complexity_measures auch '{}' und '()', die FuncDef/FuncDecl habe ich aber nicht dazugenommen

Es werden alle Operatoren mit Effekt in ein Set gespeichert und zusätzlich ein Zähler für die Gesamtanzahl inkrementiert.

Operanden:

- **ConstDef** (Konstante und das Literal dahinter))
- **VarDef** (auch **MutDef**, aber nur wenn es keine **FuncDecl/FuncDef** ist)
- Funktionsaufruf in **Fact** (Sprungmarke)
- Zeichen-Ketten für cin/cout (**InputStat | OutputStat**)

Es werden alle Operanden in ein Set gespeichert und zusätzlich ein Zähler für die Gesamtanzahl inkrementiert.

Commands

```
SG MiniC.atg
PGT MiniC.atg
csc *.cs
```

Compilation

```
C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniCpp>SG MiniCpp.atg
-------------------------------------
 C o c o - 2  SG    Version C#, 1.3
 Copyright H. Dobler     April 2020
-------------------------------------


source file in "MiniCpp.atg"
parsing ...
evaluating lexical structure ...
generating scanner ...
listing ...

compilation completed in 0:0.25

no errors detected

C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniCpp>PGT MiniCpp.atg
-------------------------------------
 C o c o - 2  PGT  Version  C#, 1.3
 Copyright H. Dobler     April 2020
-------------------------------------


source file in "MiniCpp.atg"
parsing ...
generating semantic evaluator ...
evaluating syntax ...
checking grammar ...
(162, 83): !WRN! LL(1) error (start AND succ): else
listing ...
generating syntax analyzer ...

compilation completed in 0:00.29
LL(1) conflict(s) detected

C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniCpp>csc *.cs
Microsoft (R) Visual C# Compiler version 4.4.0-6.22559.4 (d7e8a398)
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniCpp>
```

Code

```
COMPILER MiniCpp

  SEM<<

  private static int numStats = 0;
  private static int cyclomaticComplexity = 1;
  private static System.Collections.Generic.SortedSet<string> n1 = new
System.Collections.Generic.SortedSet<string>();
  private static System.Collections.Generic.SortedSet<string> n2 = new
System.Collections.Generic.SortedSet<string>();
  private static int N1 = 0;
  private static int N2 = 0;

  private static void PrintAnalysisResults() {
    System.Console.WriteLine("using .NET Framework " +
System.Environment.Version);

    System.Console.WriteLine("b)");
    System.Console.WriteLine("  Lines of Code:         " + MiniCppLex.curLine);
    System.Console.WriteLine("  Number of statements:  " + numStats);
    System.Console.WriteLine("  Cyclomatic Complexity: " + cyclomaticComplexity);

    string n1AsString = // cannot do "using System.Linq" here :(
      System.Linq.Enumerable.Aggregate(n1, (l, r) => l.ToString() + ", " +
r.ToString());
    string n2AsString =
      System.Linq.Enumerable.Aggregate(n2, (l, r) => l.ToString() + ", " +
r.ToString());
    int n = n1.Count + n2.Count;
    int N = N1 + N2;
    double V = (double)N * System.Math.Log(n, 2);
    double D = (double)(n1.Count * N2) / (double)(2 * n2.Count);
    double E = D * V;
    System.Console.WriteLine("c)");
    System.Console.WriteLine("  n1: " + n1AsString);
    System.Console.WriteLine("  n2: " + n2AsString);
    System.Console.WriteLine("  N1: " + N1);
    System.Console.WriteLine("  N2: " + N2);
    System.Console.WriteLine();
    System.Console.WriteLine("  vocabulary: " + n);
    System.Console.WriteLine("  length:     " + N);
    System.Console.WriteLine("  volume:     " + V.ToString("N2"));
    System.Console.WriteLine("  difficulty: " + D.ToString("N2"));
    System.Console.WriteLine("  effort:     " + E.ToString("N2"));
```

```
  }
  // so I won't forget to increment N1/N2 ^^
  private static void AddToN1(string item) {
    n1.Add(item); N1++;
  }

  private static void AddToN2(string item) {
    n2.Add(item); N2++;
  }

  // if user selects "[c]ontinue", we have to reset all of our collected data
  private static void Reset() {
    numStats = 0;
    cyclomaticComplexity = 1;
    n1.Clear();
    n2.Clear();
    N1 = 0;
    N2 = 0;
  }

  >>

CHARACTER SETS
  digit      = '0' .. '9'.
  whiteSpace = CHR(9) + EOL IGNORE. /*' ' ignored by default*/
  letter     = 'a' .. 'z' + 'A' .. 'Z' + '_'.
  anyButDoubleQuote = ANY - '"'.

COMMENTS
  FROM '/*' TO '*/' .
  FROM '//' TO EOL .

KEYWORDS
  'const'. 'false'. 'true'. 'nullptr'. 'bool'. 'if'. 'else'. 'while'. 'break'.
'cin'. 'cout'. 'endl'. 'delete'. 'return'. 'new'. 'void'. 'int'.

TOKENS
  '+'.  '-'.  '*'.  '/'.  '('.  ')'. '%'. '<'. '>'. '!'.
  '{'. '}'.
  ','. ';'. '='.
  '>='. '<='. '>>'. '<<'. '+='. '-='. '*='. '/='. '%='. '=='. '!='. '++'. '--'.
'||'. '&&'. '['. ']'.

TOKEN CLASSES
  number<<out int n>> =
    digit { digit }           LEX<<n = Convert.ToInt32(tokenStr);>>
  .

  ident<<out string idStr>> =
    letter { letter | digit } LEX<<idStr = tokenStr;>>
  .

  string<<out string str>> =
    '"' { anyButDoubleQuote } '"' LEX<<str = tokenStr;>>
```

```
  .

NONTERMINALS
  MiniCpp.
  ConstDef.
  Init.
  VarDef.
  FormParList.
  Type.
  Block.
  Stat.
  EmptyStat.
  BlockStat.
  ExprStat.
  IfStat.
  WhileStat.
  BreakStat.
  InputStat.
  OutputStat.
  DeleteStat.
  ReturnStat.
  Expr.
  OrExpr.
  AndExpr.
  RelExpr.
  SimpleExpr.
  Term.
  NotFact.
  Fact.
  ActParList.
  FormParTypeRight.
  MutDef.

RULES

  MiniCpp = SEM<<Reset();>> { ConstDef | MutDef | ';' }
SEM<<PrintAnalysisResults();>>
    .
  ConstDef =                          LOCAL<<string idStr1 = ""; string idStrN =
"";>>
     'const' Type ident<<out idStr1>> SEM<<AddToN2(idStr1);>> Init { ','
ident<<out idStrN>> SEM<<AddToN2(idStrN);>> Init } ';' .
  Init =                              LOCAL<<int n = 0;>>
    '=' ( false | true | 'nullptr'
    | [ '+' | '-' ] number<<out n>> SEM<<AddToN2(n.ToString());>> ) .
  // VarDef | FuncHead => not ConstDef, so its [Mut]able
  MutDef =                   LOCAL<<string idStr = "";>>
      Type [ '*' ] ident<<out idStr>> ( [ Init ] SEM<<AddToN2(idStr);>> |
       '(' [ FormParList ] ')' SEM<<AddToN1("()");>> ) [ Block ]
    .
  VarDef =                    LOCAL<<string idStr1 = ""; string idStrN = "";>>
    Type [ '*' ] ident<<out idStr1>> [ Init ] SEM<<AddToN2(idStr1);>>
    { ',' [ '*' ] ident<<out idStrN>> [ Init ] SEM<<AddToN2(idStrN);>> } ';'
    .
```

```
  FormParList =
    ( 'void' [ FormParTypeRight { ',' Type FormParTypeRight } ]
    | ('int' | 'bool' ) FormParTypeRight { ',' Type FormParTypeRight } )
    .
  FormParTypeRight =                        LOCAL<<string idStr = "";>>
    [ '*' ] ident<<out idStr>> [ '[' ']' ]
    .
  Type = 'void' | 'bool' | 'int' .
  Block = '{' { ConstDef | VarDef | Stat } '}' SEM<<AddToN1("{}");>> .
  Stat = ( EmptyStat | BlockStat | ExprStat
    | IfStat | WhileStat | BreakStat
    | InputStat | OutputStat
    | DeleteStat | ReturnStat
    ) SEM<<numStats++;>>.
  EmptyStat = ';' .
  BlockStat = Block .
  ExprStat = Expr ';' .
  IfStat = 'if' SEM<<cyclomaticComplexity++; AddToN1("if");>> '(' Expr ')' Stat [
'else' SEM<<AddToN1("else");>> Stat ] .
  WhileStat = 'while' SEM<<cyclomaticComplexity++; AddToN1("while");>> '(' Expr
')' Stat .
  BreakStat = 'break' SEM<<AddToN1("break");>> ';' .
  InputStat =                               LOCAL<<string idStr = "";>>
    'cin' SEM<<AddToN1("cin");>> '>>' ident<<out idStr>> ';' .
  OutputStat =                            LOCAL<<string str = "";>>
    'cout' SEM<<AddToN1("cout");>> '<<' ( Expr | string<<out str>>
SEM<<AddToN2(str);>>  | 'endl' )
    { '<<' ( Expr | string<<out str>> SEM<<AddToN2(str);>> | 'endl' ) } ';' .
  DeleteStat =                     LOCAL<<string idStr = "";>>
    'delete' '[' ']' ident<<out idStr>> ';' .
  ReturnStat = 'return' [ Expr ] ';' .
  Expr =
    OrExpr {
      ( '=' SEM<<AddToN1("=");>>
      | '+=' SEM<<AddToN1("+=");>>
      | '-=' SEM<<AddToN1("-=");>>
      | '*=' SEM<<AddToN1("*=");>>
      | '/=' SEM<<AddToN1("/=");>>
      | '%=' SEM<<AddToN1("%=");>> ) OrExpr } .
  OrExpr = AndExpr { '||' SEM<<AddToN1("||");>> AndExpr } .
  AndExpr = RelExpr { '&&' SEM<<AddToN1("&&");>> RelExpr } .
  RelExpr = SimpleExpr {
    ( '==' SEM<<AddToN1("==");>>
    | '!=' SEM<<AddToN1("!=");>>
    | '<' SEM<<AddToN1("<");>>
    | '<=' SEM<<AddToN1("<=");>>
    | '>' SEM<<AddToN1(">");>>
    | '>=' SEM<<AddToN1(">=");>> ) SimpleExpr } .
  SimpleExpr = [ '+' SEM<<AddToN1("+");>> | '-' SEM<<AddToN1("-");>> ]
    Term { ( '+' SEM<<AddToN1("+");>> | '-' SEM<<AddToN1("-");>> ) Term } .
  Term =
    NotFact {
      ( '*' SEM<<AddToN1("*");>>
      | '/' SEM<<AddToN1("/");>>
```

```
      | '%' SEM<<AddToN1("%");>> ) NotFact } .
  NotFact = [ '!' SEM<<AddToN1("!");>> ] Fact .
  Fact =                              LOCAL<<int n = 0; string idStr = "";>>
    ( 'false' | 'true' | 'nullptr' | number<<out n>> SEM<<AddToN1(n.ToString());>>
    | [ '++' SEM<<AddToN1("++");>> | '--' SEM<<AddToN1("--");>> ]
      ident<<out idStr>> [ ( '[' Expr ']' )
        | ( '(' [ ActParList ] ')' SEM<<AddToN2(idStr);>> )
        ]
      [ '++' SEM<<AddToN1("++");>> | '--' SEM<<AddToN1("--");>> ]
    | 'new' Type '[' Expr ']'
    | '(' Expr ')'
    ) .
  ActParList = Expr { ',' Expr } .

END MiniCpp.
```

Testfälle

Sieve.mcpp

```
C:\Users\weien\OneDrive\Documents\Studium\Master\1.
Semester\FCW1\UE05\src\MiniCpp>Main.exe
---------------------------
 MiniCpp Compiler    Version X
 Frontend gen. with Coco-2
---------------------------


source file > sieve.mcpp
parsing ...
using .NET Framework 4.0.30319.42000
b)
  Lines of Code:         72
  Number of statements:  37
  Cyclomatic Complexity: 8
c)
  n1: (), *, {}, +, ++, +=, <, <=, =, ==, >, 0, 1, 10, 2, 3, 46340, cin, cout,
else, if, while
  n2: "\t", "error: number too large (max. 46340)", "error: number too small (min.
3)", "n > ", col, i, j, n, sieve, Sieve
  N1: 59
  N2: 11

  vocabulary: 32
  length:     70
  volume:     350.00
  difficulty: 12.10
  effort:     4,235.00
no errors detected

[c]ontinue or [q]uit >
```

FizzBuzz.mcpp

```
// CPP program to print Fizz Buzz
// taken from https://www.geeksforgeeks.org/fizz-buzz-implementation/
// and modified to fit MiniCpp-language
//#include <stdio.h>

int main(void)
{
  int i;
  while (i<=100)
  {
    // number divisible by 3 and 5 will
    // always be divisible by 15, print
    // 'FizzBuzz' in place of the number
    if (i%15 == 0)
      cout << "FizzBuzz" << endl;

    // number divisible by 3? print 'Fizz'
    // in place of the number
    else if ((i%3) == 0)
      cout << "Fizz" << endl;

    // number divisible by 5, print 'Buzz'
    // in place of the number
    else if ((i%5) == 0)
      cout << "Buzz" << endl;
    else // print the Number
      cout << i << endl;
    i++;
  }

  return 0;
}
```

```
source file > FizzBuzz.mcpp
parsing ...
using .NET Framework 4.0.30319.42000
b)
  Lines of Code:         33
  Number of statements:  11
  Cyclomatic Complexity: 5
c)
  n1: %, (), {}, ++, <=, ==, 0, 100, 15, 3, 5, cout, else, if, while
  n2: "Buzz", "Fizz", "FizzBuzz", i
  N1: 30
  N2: 4
```

```
  vocabulary: 19
  length:     34
  volume:     144.43
  difficulty: 7.50
  effort:     1,083.22
no errors detected

[c]ontinue or [q]uit >
```

Test.mcpp

```
// this code is not supposed to make sense

int counter = -10;

int f2(void) {
  while (counter != -20) {
    f1();
    counter--;
  }
}

const void bla = 100;

/*
const void bla2 = 100;
*/

bool f1(int i, void j) {
  if (i % 420 == 69) { /* nice */
    counter = f2() + j * (-10);
  } else {
    if(f1(i, j / 100)) {
      counter = 0;
    }
  }
  return j > 911;
}

void main() {
  return f1(69, 21 / 5);
} // main
```

```
source file > Test.mcpp
parsing ...
using .NET Framework 4.0.30319.42000
b)
  Lines of Code:         32
  Number of statements:  13
```

```
  Cyclomatic Complexity: 4
c)
  n1: -, --, !=, %, (), *, /, {}, +, =, ==, >, 0, 10, 100, 20, 21, 420, 5, 69,
911, else, if, while
  n2: 10, 100, bla, counter, f1, f2
  N1: 37
  N2: 8

  vocabulary: 30
  length:     45
  volume:     220.81
  difficulty: 16.00
  effort:     3,532.96
no errors detected

[c]ontinue or [q]uit >
```