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.