

<input type="checkbox"/> Gr. 1, Dr. G. Kronberger	Name <u>Andreas Roither</u>	Aufwand in h <u>6</u> h
<input type="checkbox"/> Gr. 2, Dr. H. Gruber		
<input checked="" type="checkbox"/> Gr. 3, Dr. D. Auer	Punkte _____	Kurzzeichen Tutor / Übungsleiter _____ / _____

## 1. Transformation arithmetischer Ausdrücke

(4 + 6 Punkte)

Wie Sie wissen, können einfache arithmetische Ausdrücke in der Infix-Notation, z. B.  $(a + b) * c$ , durch folgende Grammatik beschrieben werden:

Expr = Term { '+' Term | '-' Term } .  
Term = Fact { '\*' Fact | '/' Fact } .  
Fact = number | ident | '(' Expr ')' .

Die folgende attributierte Grammatik (ATG) beschreibt die Transformation einfacher arithmetischer Ausdrücke von der Infix- in die Postfix-Notation, z. B. von  $(a + b) * c$  nach  $a b + c *$ .

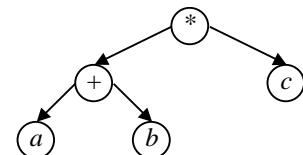
Expr =	Term =	Fact =
Term	Fact	number $\uparrow_n$ sem Write(n); endsem
{ '+' Term sem Write('+'); endsem	{ '*' Fact sem Write('*'); endsem	ident $\uparrow_{id}$ sem Write(id); endsem
'-' Term sem Write('-'); endsem	'/' Fact sem Write('/'); endsem	'(' Expr ')' .
} .	} .	

- Entwickeln Sie eine ATG zur Transformation einfacher arithmetischer Ausdrücke von der Infix- in die Präfix-Notation, also z. B. von  $(a + b) * c$  nach  $* + a b c$ .
- Implementieren Sie die ATG aus a) und testen Sie Ihre Implementierung ausführlich.

## 2. Arithmetische Ausdrücke und Binärbäume

(4 + 6 + 1 + 3 Punkte)

Arithmetische Ausdrücke können im Hauptspeicher auch in Form von Binärbäumen dargestellt werden. Z. B. entspricht dem Infix-Ausdruck  $(a + b) * c$  der rechts dargestellte Binärbaum.



- Entwickeln Sie eine ATG, die arithmetische Infix-Ausdrücke in Binärbäume (gemäß der Deklarationen unten) umwandelt.

```

TYPE
  NodePtr = Node;
  Node = RECORD
    left, right: NodePtr;
    txt: STRING, (*operator or operand, both in textual representation*)
  END; (*Node*)
  TreePtr = NodePtr;

```

- Implementieren Sie die ATG aus a) und testen Sie Ihre Implementierung ausführlich.
- Geben Sie die Ergebnisbäume durch entsprechende Baumdurchläufe *in-order*, *pre-order* und *post-order* aus: Was stellen Sie dabei fest?
- Implementieren Sie eine rekursive Funktion

```

FUNCTION ValueOf (t: TreePtr): INTEGER;

```

die den Baum "auswertet", also den Wert des Ausdrucks berechnet, der durch den Baum repräsentiert wird. (Hinweis: In einem *post-order*-Baumdurchlauf zuerst den Wert des linken Unterbaums, dann den Wert des rechten Unterbaums berechnen und zum Schluss in Abhängigkeit vom Operator in der Wurzel den Gesamtwert berechnen).

# Übung 5

## Aufgabe 1

### Lösungsidee

Es wird eine ATG für Infix zu Prefix erstellt. Mithilfe dieser ATG wird ein entsprechende Implementation vorgenommen. Um eine Prefix-Notation zu erreichen wird Auf oberster Ebene ( Expr ) alles aus den unteren Ebene ( bzw. aus den Aufrufen von Term und Fact ) aneinander gehängt. Somit wird eine Prefix-Notation erreicht.

```

1  S <out String result> =
    Expr <out e>                sem reslut := e; endsem
3  eos.

5  Expr <out String e> =
    Term <out t1>                sem e := t1; endsem
7  { '+' Term <out t2>          sem e := ' + ' + t1 + ' ' + t2; t1 := e;
    endsem
    | '-' Term <out t2>         sem e := ' - ' + t1 + ' ' + t2; t1 := e;
    endsem
9  }.

11 Term <out String t> =
    Fact <out f1>                sem t := f1; endsem
13 { '*' Fact <out f2>          sem t := ' * ' + f1 + ' ' + f2; f1 := t;
    endsem
    | '/' Fact <out f2>         sem t := ' / ' + f1 + ' ' + f2; f1 := t;
    endsem
15 }.

17 Fact <out String f> =
    number <out stringVal>       sem f := numberVal endsem
19 | variable <out id>          sem f := variableStr; endsem
    | '(' Expr <out e>          sem f := e; endsem
21 | ')' .

```

InfixToPrefixATG.txt

Die ATG für Infix zu Prefix.

```
1  (* InfixToPrefix    26.04.17 *)
3  PROGRAM InfixToPrefix;
   CONST
5     eosCh = Chr(0);

7  TYPE
   SymbolCode = (noSy, (* error symbol *)
9                 eosSy,
11                plusSy, minusSy, timesSy, divSy,
13                leftParSy, righParSy,
15                number, variable);

17  VAR
   line: STRING;

19  ch: CHAR;
   cnr: INTEGER;
   sy: SymbolCode;
   numberVal, variableStr: STRING;
   success: BOOLEAN;

21  (* ===== Scanner ===== *)
23  PROCEDURE NewCh;
   BEGIN
25     IF cnr < Length(line) THEN BEGIN
27         cnr := cnr + 1;
29         ch := line[cnr];
31     END
33     ELSE BEGIN
35         ch := eosCh;
37     END;
39  END;

41  PROCEDURE NewSy;
   VAR
43     numberStr: STRING;
45     code: INTEGER;
47  BEGIN
   WHILE ch = ' ' DO BEGIN
       NewCh;
   END;

   CASE ch OF
       eosCh: BEGIN
           sy := eosSy;
       END;
       '+': BEGIN
```

```

    sy := plusSy;
    NewCh;
49  END;
51  '-' : BEGIN
    sy := minusSy;
53  NewCh;
    END;
55  '*' : BEGIN
    sy := timesSy;
57  NewCh;
    END;
59  '/' : BEGIN
    sy := divSy;
61  NewCh;
    END;
63  '(' : BEGIN
    sy := leftParSy;
65  NewCh;
    END;
67  ')' : BEGIN
    sy := righParSy;
69  NewCh;
    END;
71  (* for numbers *)
    '0'..'9' : BEGIN
73  sy := number;
    numberStr := '';
75
    WHILE (ch >= '0') AND (ch <= '9') DO BEGIN
77  numberStr := numberStr + ch;
    NewCh;
79  END;
    numberVal := numberStr;
81  END;

83  (* for characters *)
    'A' .. 'Z', 'a'..'z' : BEGIN
85  sy := variable;
    variableStr := '';
87
    WHILE ((ch >= 'A') AND (ch < 'Z')) OR ((ch >= 'a') AND (ch < 'z'))
89  DO BEGIN
    variableStr := variableStr + ch;
91  NewCh;
    END;
93  END;
ELSE
95  sy := noSy;
```

```

    END;
97  END;

99  (* ===== PARSER ===== *)
PROCEDURE S; FORWARD;
101 PROCEDURE Expr(VAR e: STRING); FORWARD;
PROCEDURE Term(VAR t: STRING); FORWARD;
103 PROCEDURE Fact(VAR f: STRING); FORWARD;

105 PROCEDURE S;
VAR
107   e: STRING;
BEGIN
109   Expr(e); IF NOT success THEN EXIT;
    (* SEM *)
111   WriteLn('result= ', e);
    (* ENDSEM *)
113   IF sy <> eosSy THEN BEGIN success := FALSE; EXIT; END;
END;
115

PROCEDURE Expr(VAR e: STRING);
117   VAR
    t1, t2: STRING;
119   BEGIN
    Term(t1); IF NOT success THEN EXIT;
121    (* SEM *)
    e := t1;
123    (* ENDSEM *)
    WHILE (sy = plusSy) OR (sy = minusSy) DO BEGIN
125      CASE sy OF
        plusSy: BEGIN
127          NewSy;
          Term(t2); IF NOT success THEN EXIT;
129          (* SEM *)
          e := ' + ' + t1 + ' ' + t2;
131          t1 := e;
          (* ENDSEM *)
133        END;
        minusSy: BEGIN
135          NewSy;
          Term(t2); IF NOT success THEN EXIT;
137          (* SEM *)
          e := ' - ' + t1 + ' ' + t2;
139          t1 := e;
          (* ENDSEM *)
141        END;
      END;
143   END;
END;

```

```

END;
145
PROCEDURE Term(VAR t: STRING);
147   VAR
      f1, f2: STRING;
149 BEGIN
    Fact(f1); IF NOT success THEN EXIT;
151    (* SEM *)
    t := f1;
153    (* ENDSEM *)
    WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN
155      CASE sy OF
        timesSy: BEGIN
157          NewSy;
          Fact(f2); IF NOT success THEN EXIT;
159          (* SEM *)
          t := ' * ' + f1 + ' ' + f2;
161          f1 := t;
          (* ENDSEM *)
163        END;
        divSy: BEGIN
165          NewSy;
          Fact(f2); IF NOT success THEN EXIT;
167          (* SEM *)
          t := ' / ' + f1 + ' ' + f2;
169          f1 := t;
          (* ENDSEM *)
171        END;
      END;
173    END;
END;
175
PROCEDURE Fact(VAR f: STRING);
177 BEGIN
    CASE sy OF
179      number: BEGIN
          (* SEM *)
181          f := numberVal;
          (* ENDSEM *)
183          NewSy;
          END;
185      variable: BEGIN
          f := variableStr;
187          NewSy;
          END;
189      leftParSy: BEGIN
          NewSy;
191          Expr(f); IF NOT success THEN EXIT;

```


```
193         IF sy <> righParSy THEN BEGIN success:= FALSE; EXIT; END;
        NewSy;
195     END;
    ELSE
197         success := FALSE;
    END;
199 (* ===== END PARSER ===== *)

201 PROCEDURE SyntaxTest(str: STRING);
    BEGIN
203     WriteLn('Infix: ', str);
        line := str;
205     cnr := 0;
        NewCh;
207     NewSy;
        success := TRUE;

209     S;
211     IF success THEN WriteLn('successful syntax analysis',#13#10)
        ELSE WriteLn('Error in column: ', cnr,#13#10);
213 END;

215 BEGIN
    (* Test cases *)
217     SyntaxTest('(a + b) * c');
        SyntaxTest('1 + 2 + 3');
219     SyntaxTest('(1 + 2) * a');
        SyntaxTest('((a + b) * c)');
221     SyntaxTest('a++3*4');
        SyntaxTest('1+2+3+4+5+6+7+8');
223 END.
```

InfixToPrefix.pas



```
C:\windows\system32\cmd.exe

C:\Users\andir\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 5>InfixToPrefix.exe
Infix: (a + b) * c
result= * + a b c
successful syntax analysis

Infix: 1 + 2 + 3
result= + + 1 2 3
successful syntax analysis

Infix: (1 + 2) * a
result= * + 1 2 a
successful syntax analysis

Infix: ((a + b) * c)
Error in column: 14

Infix: a++3*4
Error in column: 4

Infix: 1+2+3+4+5+6+7+8
result= + + + + + + + 1 2 3 4 5 6 7 8
successful syntax analysis

C:\Users\andir\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 5>
```

Abbildung 1: Infix to Prefix Test

Die Testfälle zeigen sowohl funktionierende Testfälle als auch Testfälle mit eingebauten Fehlern. Falls zu viele Klammern oder Rechenoperationszeichen übergeben werden, wird eine Fehler Meldung ausgegeben. Die Spalten Nummer bei der Fehler Meldung funktioniert dabei leider nicht immer. Result zeigt die Prefix-Notation.



## Aufgabe 2

### Lösungsidee

Die ATG funktioniert ähnlich wie bei Aufgabe 1. Der Unterschied besteht darin das der Baum ohne eine Insert Funktion aufgebaut wird. Die Nodes werden aneinander gehängt und somit wird der Baum aufgebaut. Auf oberster Ebene ( oberster Funktionsaufruf, oder erste Funktion die aufgerufen wird von Expr, Term, Fact ) wird eine Node mit den anderen Nodes aus den Funktionsaufruf Term aneinander gehängt. Bei "1 + 2" wäre f1 eine Node mit "1" in txt und f2 eine Node mit "2" in txt gespeichert. Mit diesen Nodes wird eine neue Node erstellt, mit "+" als Wurzelknoten und f1, f2 als die beiden sub trees. Die anderen Funktionen geben immer eine Node zurück, entweder mit einem "+", "-", oder einer Zahl als Wurzelknoten. Die rekursive Funktion wird mithilfe eines case statements implementiert. Je nachdem welches Zeichen in der aktuellen Node enthalten ist wird eine der vier Rechenoperationen ausgeführt. Bei den verschiedene Ausgaben InOrder, PreOrder, PostOrder fällt auf das InOrder den Baum ähnlich ausgibt wie den ursprünglichen Input nur ohne Klammern, PreOrder gibt den Baum aus wie Prefix-Notation und PostOrder wie Postfix-Notation.

```

1  S <out Node result> =
2      Expr <out e>                sem reslut := e; endsem
3      eos.
4
5  Expr <out Node e> =
6      Term <out t1>                sem e := t1; endsem
7      { '+' Term <out t2>          sem e := NewNode(t1,t2,'+'); t1 := e; endsem
8      | '-' Term <out t2>          sem e := NewNode(t1,t2,'-'); t1 := e; endsem
9      }.
10
11 Term <out Node t> =
12 Fact <out f1>                    sem t := f1; endsem
13 { '*' Fact <out f2>              sem t := NewNode(f1,f2,'*'); f1 := t; endsem
14 | '/' Fact <out f2>              sem t := NewNode(f1,f2,'/'); f1 := t; endsem
15 }.
16
17 Fact <out Node f> =
18 number <out stringVal>          sem f := NewNode(numberVal); endsem
19 | variable <out id>
20
21 sem f := NewNode(variableStr); endsem
22 | '(' Expr <out e>                sem f := e; endsem
23 | ')'.

```

TreeATG.txt

```
1  (* TreeEval      26.04.17 *)
3  PROGRAM TreeEval;
   CONST
5     eosCh = Chr(0);

7  TYPE
   SymbolCode = (noSy, (* error symbol *)
9                 eosSy,
                plusSy, minusSy, timesSy, divSy,
11                leftParSy, righParSy,
                number, variable);

13
   NodePtr = ^Node;
15   Node = RECORD
       txt: STRING;
17       left, right: NodePtr;
   END;
19   TreePtr = NodePtr;

21  VAR
   sy: SymbolCode;
23   ch: CHAR;
   cnr: INTEGER;
25   numberVal, variableStr, line: STRING;
   success: BOOLEAN;
27   tr : TreePtr;

29  (* ===== Scanner ===== *)
   PROCEDURE NewCh;
31  BEGIN
       IF cnr < Length(line) THEN BEGIN
33         cnr := cnr + 1;
         ch := line[cnr];
35       END
       ELSE BEGIN
37         ch := eosCh;
       END;
39  END;

41  PROCEDURE NewSy;
   VAR
43     numberStr: STRING;
     code: INTEGER;
45  BEGIN
       WHILE ch = ' ' DO BEGIN
47         NewCh;
```

```

END;
49
CASE ch OF
51   eosCh: BEGIN
        sy := eosSy;
53   END;
        '+' : BEGIN
55         sy := plusSy;
        NewCh;
57   END;
        '-' : BEGIN
59         sy := minusSy;
        NewCh;
61   END;
        '*' : BEGIN
63         sy := timesSy;
        NewCh;
65   END;
        '/' : BEGIN
67         sy := divSy;
        NewCh;
69   END;
        '(' : BEGIN
71         sy := leftParSy;
        NewCh;
73   END;
        ')' : BEGIN
75         sy := righParSy;
        NewCh;
77   END;
        (* for numbers *)
79   '0'..'9': BEGIN
        sy := number;
81     numberStr := '';

83     WHILE (ch >= '0') AND (ch <= '9') DO BEGIN
        numberStr := numberStr + ch;
85     NewCh;
87     END;
        numberVal := numberStr;
89   END;

        (* for characters *)
91   'A' .. 'Z', 'a'..'z': BEGIN
        sy := variable;
93     variableStr := '';

95     WHILE ((ch >= 'A') AND (ch < 'Z')) OR ((ch >= 'a') AND (ch < 'z'))

```

```

DO BEGIN
97     variableStr := variableStr + ch;
        NewCh;
99     END;
    END;
101    ELSE
        sy := noSy;
103    END;
END;
105

(* ===== PARSER ===== *)
107 PROCEDURE S; FORWARD;
PROCEDURE Expr(VAR e: NodePtr); FORWARD;
109 PROCEDURE Term(VAR t: NodePtr); FORWARD;
PROCEDURE Fact(VAR f: NodePtr); FORWARD;
111 FUNCTION NewNode (value: STRING): NodePtr; FORWARD;
FUNCTION NewNode2 (leftSubTree, rightSubTree: NodePtr; value: STRING)
113     : NodePtr; FORWARD;

115 PROCEDURE S;
VAR
117     e: NodePtr;
BEGIN
119     Expr(e); IF NOT success THEN EXIT;
        tr := e;
121     IF sy <> eosSy THEN BEGIN success := FALSE; EXIT; END;
END;
123

PROCEDURE Expr(VAR e: NodePtr);
125     VAR
        t1, t2: NodePtr;
127     BEGIN
        Term(t1); IF NOT success THEN EXIT;
129         (* SEM *)
        e := t1;
131         (* ENDSEM *)
        WHILE (sy = plusSy) OR (sy = minusSy) DO BEGIN
133             CASE sy OF
                plusSy: BEGIN
135                     NewSy;
                        Term(t2); IF NOT success THEN EXIT;
137                         (* SEM *)
                        e := NewNode2(t1, t2, '+');
139                         t1 := e;
                        (* ENDSEM *)
141                     END;
                minusSy: BEGIN
143                     NewSy;

```

```

145         Term(t2); IF NOT success THEN EXIT;
146         (* SEM *)
147         e := NewNode2(t1, t2, '-');
148         t1 := e;
149         (* ENDSEM *)
150     END;
151 END;
152 END;
153
154 PROCEDURE Term(VAR t: NodePtr);
155     VAR
156         f1, f2: NodePtr;
157     BEGIN
158         Fact(f1); IF NOT success THEN EXIT;
159         (* SEM *)
160         t := f1;
161         (* ENDSEM *)
162         WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN
163             CASE sy OF
164                 timesSy: BEGIN
165                     NewSy;
166                     Fact(f2); IF NOT success THEN EXIT;
167                     (* SEM *)
168                     t := NewNode2(f1, f2, '*');
169                     f1 := t;
170                     (* ENDSEM *)
171                 END;
172                 divSy: BEGIN
173                     NewSy;
174                     Fact(f2); IF NOT success THEN EXIT;
175                     (* SEM *)
176                     t := NewNode2(f1, f2, '/');
177                     f1 := t;
178                     (* ENDSEM *)
179                 END;
180             END;
181         END;
182     END;
183
184 PROCEDURE Fact(VAR f: NodePtr);
185     BEGIN
186         CASE sy OF
187             number: BEGIN
188                 (* SEM *)
189                 f := NewNode(numberVal);
190                 (* ENDSEM *)
191                 NewSy;

```

```

        END;
193   variable: BEGIN
        f := NewNode(variableStr);
195       NewSy;
        END;
197   leftParSy: BEGIN
        NewSy;
199       Expr(f); IF NOT success THEN EXIT;
        IF sy <> righParSy THEN BEGIN success:= FALSE; EXIT; END;
201       NewSy;
        END;
203   ELSE
        success := FALSE;
205   END;
END;
207 (* ===== END PARSER ===== *)

209 (* ===== TREE ===== *)

211 PROCEDURE InitTree (VAR t: TreePtr);
BEGIN
213   t:= NIL;
END;

215 (* NewNode with string *)
217 FUNCTION NewNode (value: STRING): NodePtr;
VAR
219   n: NodePtr;
BEGIN
221   New(n);
   n^.txt := value;
223   n^.left := NIL;
   n^.right := NIL;
225   NewNode := n;
END;

227 (* NewNode2 with value as root and the two other
229   Nodes as left and right subtree *)
FUNCTION NewNode2 (leftSubTree, rightSubTree: NodePtr; value: STRING)
231   : NodePtr;
VAR
233   n: NodePtr;
BEGIN
235   New(n);
   n^.txt := value;
237   n^.left := leftSubTree;
   n^.right := rightSubTree;
239   NewNode2 := n;

```

```
END;
241
PROCEDURE WriteTreeInOrder (t: TreePtr);
243 BEGIN
    IF t <> NIL THEN BEGIN
245        WriteTreeInOrder(t^.left);
        Write(t^.txt);
247        WriteTreeInOrder(t^.right);
    END;
249 END;

PROCEDURE WriteTreePreOrder (t: TreePtr);
251 BEGIN
    IF t <> NIL THEN BEGIN
253        Write(t^.txt);
        WriteTreePreOrder(t^.left);
255        WriteTreePreOrder(t^.right);
    END;
257 END;

PROCEDURE WriteTreePostOrder (t: TreePtr);
261 BEGIN
    IF t <> NIL THEN BEGIN
263        WriteTreePostOrder(t^.left);
        WriteTreePostOrder(t^.right);
265        Write(t^.txt);
    END;
267 END;

(* calculate value of the tree *)
FUNCTION ValueOf(t: TreePtr): INTEGER;
271 BEGIN
    IF t <> NIL THEN BEGIN
273        CASE t^.txt OF
            '+': BEGIN
275                ValueOf := ValueOf(t^.left) + ValueOf(t^.right);
            END;
            '-': BEGIN
277                ValueOf := ValueOf(t^.left) - ValueOf(t^.right);
            END;
            '*': BEGIN
279                ValueOf := ValueOf(t^.left) * ValueOf(t^.right);
            END;
            '/': BEGIN
283                ValueOf := ValueOf(t^.left) DIV ValueOf(t^.right);
            END;
285        ELSE BEGIN
287            Val(t^.txt, ValueOf);
```

```

        END;
289     END;
        END;
291     END;

293     PROCEDURE DisposeTree(VAR t: TreePtr);
        BEGIN
295         IF t <> NIL THEN BEGIN
            DisposeTree(t^.left);
297             DisposeTree(t^.right);
            Dispose(t);
299             t := NIL;
        END;
301     END;

303     (* ===== END TREE ===== *)

305     PROCEDURE SyntaxTest(str: STRING);
        BEGIN
307         WriteLn('Infix: ', str);
            line := str;
309         cnr := 0;
            NewCh;
311         NewSy;
            success := TRUE;

313         S;

315         IF success THEN WriteLn('successful syntax analysis',#13#10)
            ELSE WriteLn('Error in column: ', cnr,#13#10);

317         WriteLn('-InOrder-');
            WriteTreeInOrder(tr);
            WriteLn(#13#10, '-PreOrder-');
321             WriteTreePreOrder(tr);
            WriteLn(#13#10, '-PostOrder-');
323             WriteTreePostOrder(tr);
            WriteLn(#13#10, 'ValueOf: ', ValueOf(tr), #13#10);
325     END;

327 BEGIN
    InitTree(tr);
329     SyntaxTest('(1 + 2) * 3');
    DisposeTree(tr);

331     InitTree(tr);
333     SyntaxTest('(1 + 2) * 3 + 2');
    DisposeTree(tr);

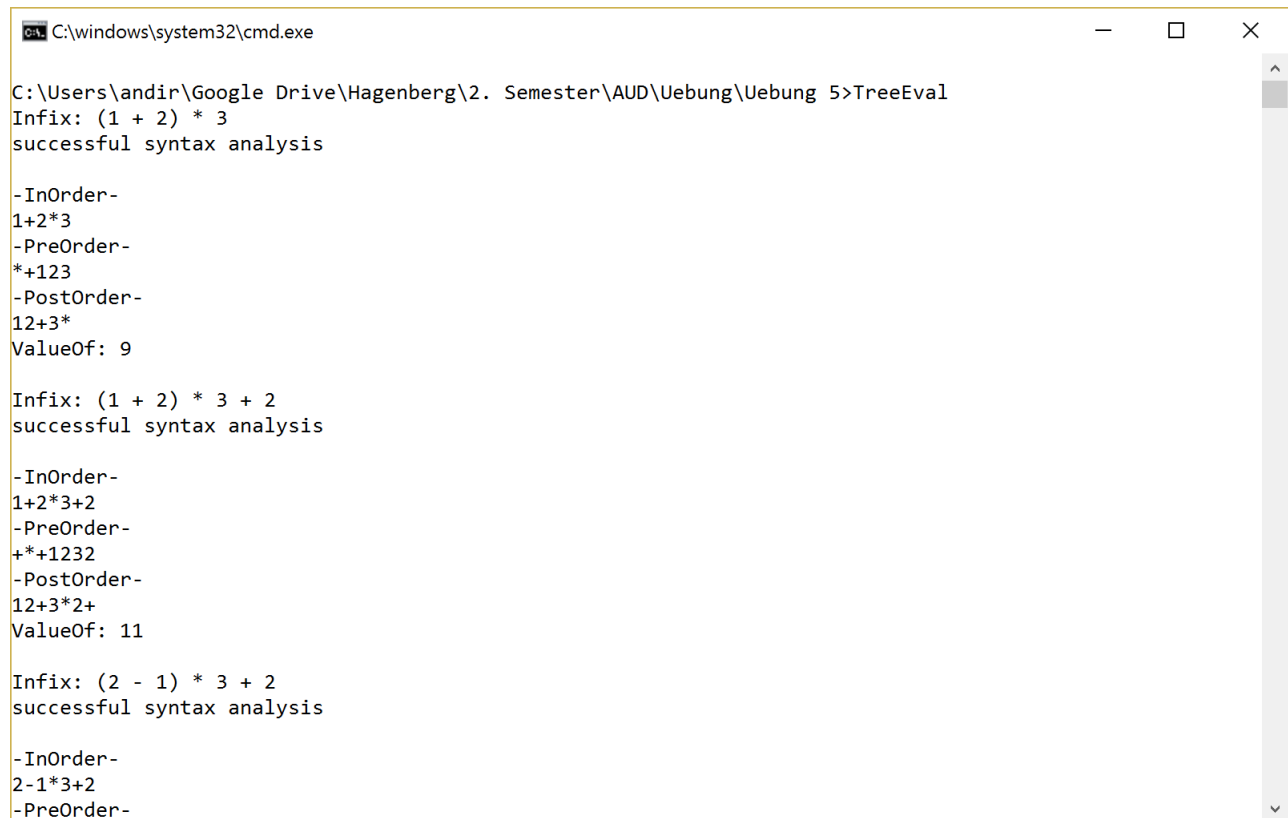
335
```



```
337 InitTree(tr);
    SyntaxTest('(2 - 1) * 3 + 2');
    DisposeTree(tr);

339
    InitTree(tr);
341 SyntaxTest('(1 + (2*4)) * 2');
    DisposeTree(tr);
343 END.
```

TreeEval.pas



```
C:\windows\system32\cmd.exe

C:\Users\andir\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 5>TreeEval
Infix: (1 + 2) * 3
successful syntax analysis

-InOrder-
1+2*3
-PreOrder-
*+123
-PostOrder-
12+3*
ValueOf: 9

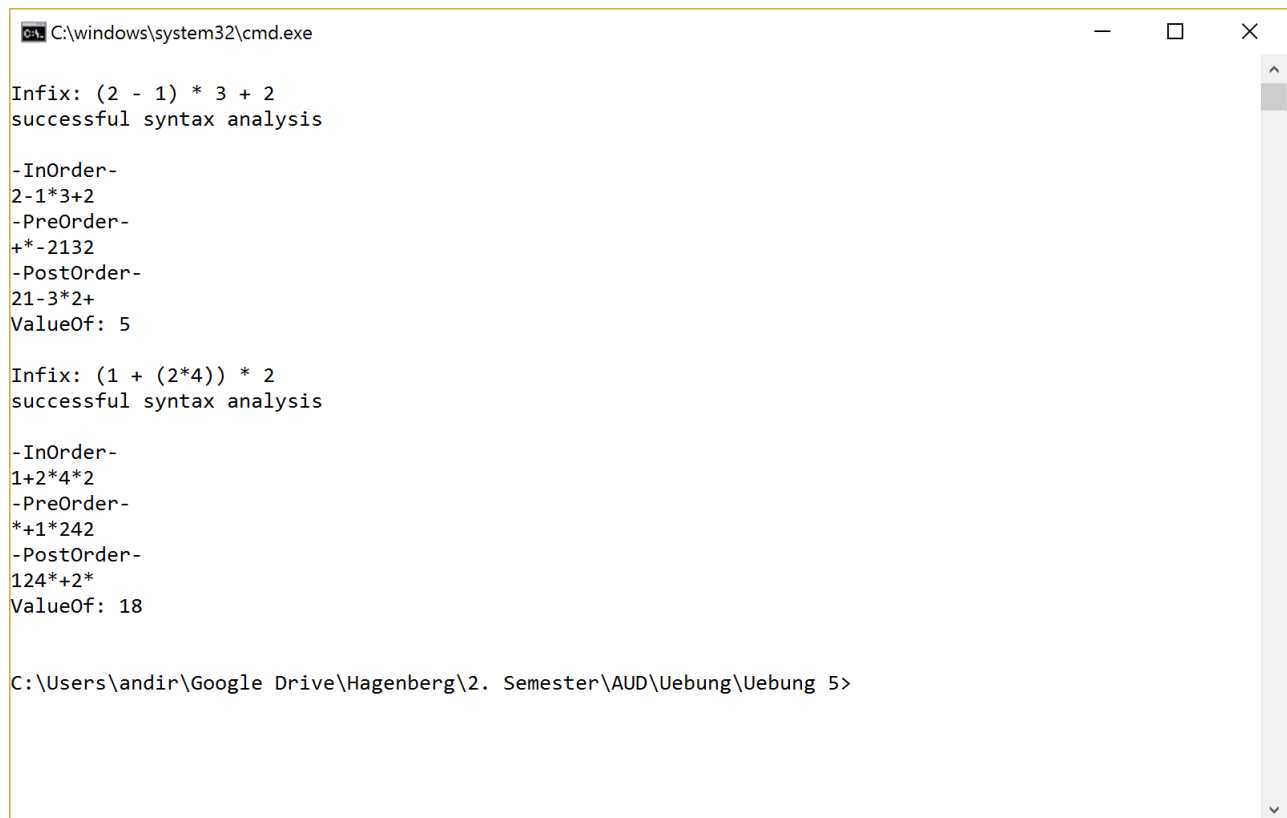
Infix: (1 + 2) * 3 + 2
successful syntax analysis

-InOrder-
1+2*3+2
-PreOrder-
+*+1232
-PostOrder-
12+3*2+
ValueOf: 11

Infix: (2 - 1) * 3 + 2
successful syntax analysis

-InOrder-
2-1*3+2
-PreOrder-
```

Abbildung 2: TreeEval Test 1



```
C:\windows\system32\cmd.exe

Infix: (2 - 1) * 3 + 2
successful syntax analysis

-InOrder-
2-1*3+2
-PreOrder-
+*-2132
-PostOrder-
21-3*2+
ValueOf: 5

Infix: (1 + (2*4)) * 2
successful syntax analysis

-InOrder-
1+2*4*2
-PreOrder-
*+1*242
-PostOrder-
124*+2*
ValueOf: 18

C:\Users\andir\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 5>
```

Abbildung 3: TreeEval Test 2

Die verschiedene Testfälle zeigen die Syntax Analysis, InOrder, PreOrder, PostOrder Ausgabe des Baumes und das verwenden der ValueOf Funktion. Die ValueOf Funktion führt alle Rechenoperationen im Baum aus und liefert das ausgerechnete Ergebnis zurück.