

<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  $a b + c *$ .

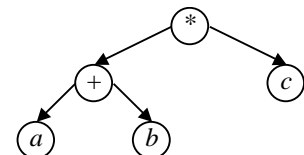
Expr =	Term =	Fact =
Term	Fact	number $\uparrow_n$ sem Write(n); endsem
{ '+' Term sem Write('+'); endsem	{ '*' Fact sem Write('*'); endsem	ident $\uparrow_{id}$ sem Write(id); endsem
'-' Term sem Write('-'); endsem	'/' Fact sem Write('/'); endsem	'(' Expr ')' .
} .	} .	

- 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;
4  CONST
5      eosCh = Chr(0);
7  TYPE
8      SymbolCode = (noSy, (* error symbol *)
9                      eosSy,
10                     plusSy, minusSy, timesSy, divSy,
11                     leftParSy, righParSy,
12                     number, variable);
13  VAR
14      line: STRING;
15
16      ch: CHAR;
17      cnr: INTEGER;
18      sy: SymbolCode;
19      numberVal, variableStr: STRING;
20      success: BOOLEAN;
21
22  (* ===== Scanner ===== *)
23  PROCEDURE NewCh;
24  BEGIN
25      IF cnr < Length(line) THEN BEGIN
26          cnr := cnr + 1;
27          ch := line[cnr];
28      END
29      ELSE BEGIN
30          ch := eosCh;
31      END;
32  END;
33
34  PROCEDURE NewSy;
35  VAR
36      numberStr: STRING;
37      code: INTEGER;
38  BEGIN
39      WHILE ch = ' ' DO BEGIN
40          NewCh;
41      END;
42
43      CASE ch OF
44          eosCh: BEGIN
45              sy := eosSy;
46              END;
47          '+': 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')) DO
89  BEGIN
    variableStr := variableStr + ch;
91  NewCh;
    END;
93  END;
ELSE
95  sy := noSy;
```

```

    END;
97  END;

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

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);
    VAR
117       t1, t2: STRING;
BEGIN
119   Term(t1); IF NOT success THEN EXIT;
    (* SEM *)
121   e := t1;
    (* ENDSEM *)
123   WHILE (sy = plusSy) OR (sy = minusSy) DO BEGIN
    CASE sy OF
125       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;

PROCEDURE Term(VAR t: STRING);

VAR

f1, f2: STRING;

BEGIN

Fact(f1); IF NOT success THEN EXIT;

(\* SEM \*)

t := f1;

(\* ENDSEM \*)

WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN

CASE sy OF

timesSy: BEGIN

NewSy;

Fact(f2); IF NOT success THEN EXIT;

(\* SEM \*)

t := ' \* ' + f1 + ' ' + f2;

f1 := t;

(\* ENDSEM \*)

END;

divSy: BEGIN

NewSy;

Fact(f2); IF NOT success THEN EXIT;

(\* SEM \*)

t := ' / ' + f1 + ' ' + f2;

f1 := t;

(\* ENDSEM \*)

END;

END;

END;

END;

PROCEDURE Fact(VAR f: STRING);

BEGIN

CASE sy OF

number: BEGIN

(\* SEM \*)

f := numberVal;

(\* ENDSEM \*)

NewSy;

END;

variable: BEGIN

f := variableStr;

NewSy;

END;

leftParSy: BEGIN

NewSy;

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;
  (* ===== 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);
      END;

213  BEGIN
215      (* Test cases *)
      SyntaxTest('(a + b) * c');
217      SyntaxTest('1 + 2 + 3');
      SyntaxTest('(1 + 2) * a');
219      SyntaxTest('((a + b) * c)');
      SyntaxTest('a++3*4');
221      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> =
    Expr <out e>          sem reslut := e; endsem
3  eos.

5  Expr <out Node e> =
    Term <out t1>          sem e := t1; endsem
7  { '+' Term <out t2>      sem e := NewNode(t1,t2,'+'); t1 := e; endsem
    | '-' Term <out t2>      sem e := NewNode(t1,t2,'-'); t1 := e; endsem
9  }.

11 Term <out Node t> =
    Fact <out f1>          sem t := f1; endsem
13 { '*' Fact <out f2>      sem t := NewNode(f1,f2,'*'); f1 := t; endsem
    | '/' Fact <out f2>      sem t := NewNode(f1,f2,'/'); f1 := t; endsem
15 }.

17 Fact <out Node f> =
    number <out stringVal> sem f := NewNode(numberVal); endsem
19 | variable <out id>

21 sem f := NewNode(variableStr); endsem
    | '(' Expr <out e>      sem f := e; endsem
23 ')'.

```

TreeATG.txt

```
1  (* TreeEval    26.04.17 *)
3  PROGRAM TreeEval;
4  CONST
5      eosCh = Chr(0);
7  TYPE
8      SymbolCode = (noSy, (* error symbol *)
9                      eosSy,
10                     plusSy, minusSy, timesSy, divSy,
11                     leftParSy, righParSy,
12                     number, variable);
13
14     NodePtr = ^Node;
15     Node = RECORD
16         txt: STRING;
17         left, right: NodePtr;
18     END;
19     TreePtr = NodePtr;
21
22     VAR
23         sy: SymbolCode;
24         ch: CHAR;
25         cnr: INTEGER;
26         numberVal, variableStr, line: STRING;
27         success: BOOLEAN;
28         tr : TreePtr;
29
30     (* ===== Scanner ===== *)
31     PROCEDURE NewCh;
32     BEGIN
33         IF cnr < Length(line) THEN BEGIN
34             cnr := cnr + 1;
35             ch := line[cnr];
36         END
37         ELSE BEGIN
38             ch := eosCh;
39         END;
40     END;
41
42     PROCEDURE NewSy;
43     VAR
44         numberStr: STRING;
45         code: INTEGER;
46     BEGIN
47         WHILE ch = ' ' DO BEGIN
48             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;
        END;
87   numberVal := numberStr;
        END;

89
(* 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): NodePtr;
    FORWARD;
113
PROCEDURE S;
115 VAR
    e: NodePtr;
117 BEGIN
    Expr(e); IF NOT success THEN EXIT;
119    tr := e;
    IF sy <> eosSy THEN BEGIN success := FALSE; EXIT; END;
121 END;

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

```

```

143     Term(t2); IF NOT success THEN EXIT;
        (* SEM *)
145     e := NewNode2(t1, t2, '-');
        t1 := e;
147     (* ENDSEM *)
        END;
149     END;
        END;
151 END;

153 PROCEDURE Term(VAR t: NodePtr);
    VAR
155     f1, f2: NodePtr;
    BEGIN
157     Fact(f1); IF NOT success THEN EXIT;
        (* SEM *)
159     t := f1;
        (* ENDSEM *)
161     WHILE (sy = timesSy) OR (sy = divSy) DO BEGIN
        CASE sy OF
163         timesSy: BEGIN
            NewSy;
165             Fact(f2); IF NOT success THEN EXIT;
                (* SEM *)
167             t := NewNode2(f1, f2, '*');
                f1 := t;
169             (* ENDSEM *)
            END;
171         divSy: BEGIN
            NewSy;
173             Fact(f2); IF NOT success THEN EXIT;
                (* SEM *)
175             t := NewNode2(f1, f2, '/');
                f1 := t;
177             (* ENDSEM *)
            END;
179         END;
        END;
181 END;

183 PROCEDURE Fact(VAR f: NodePtr);
    BEGIN
185     CASE sy OF
        number: BEGIN
187         (* SEM *)
            f := NewNode(numberVal);
189         (* ENDSEM *)
            NewSy;

```

```

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

207
(* ===== TREE ===== *)

209
PROCEDURE InitTree (VAR t: TreePtr);
211 BEGIN
    t:= NIL;
213 END;

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

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

```

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

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

255  PROCEDURE WriteTreePostOrder (t: TreePtr);
      BEGIN
257    IF t <> NIL THEN BEGIN
          WriteTreePostOrder(t^.left);
259    WriteTreePostOrder(t^.right);
          Write(t^.txt);
261    END;
      END;

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

```

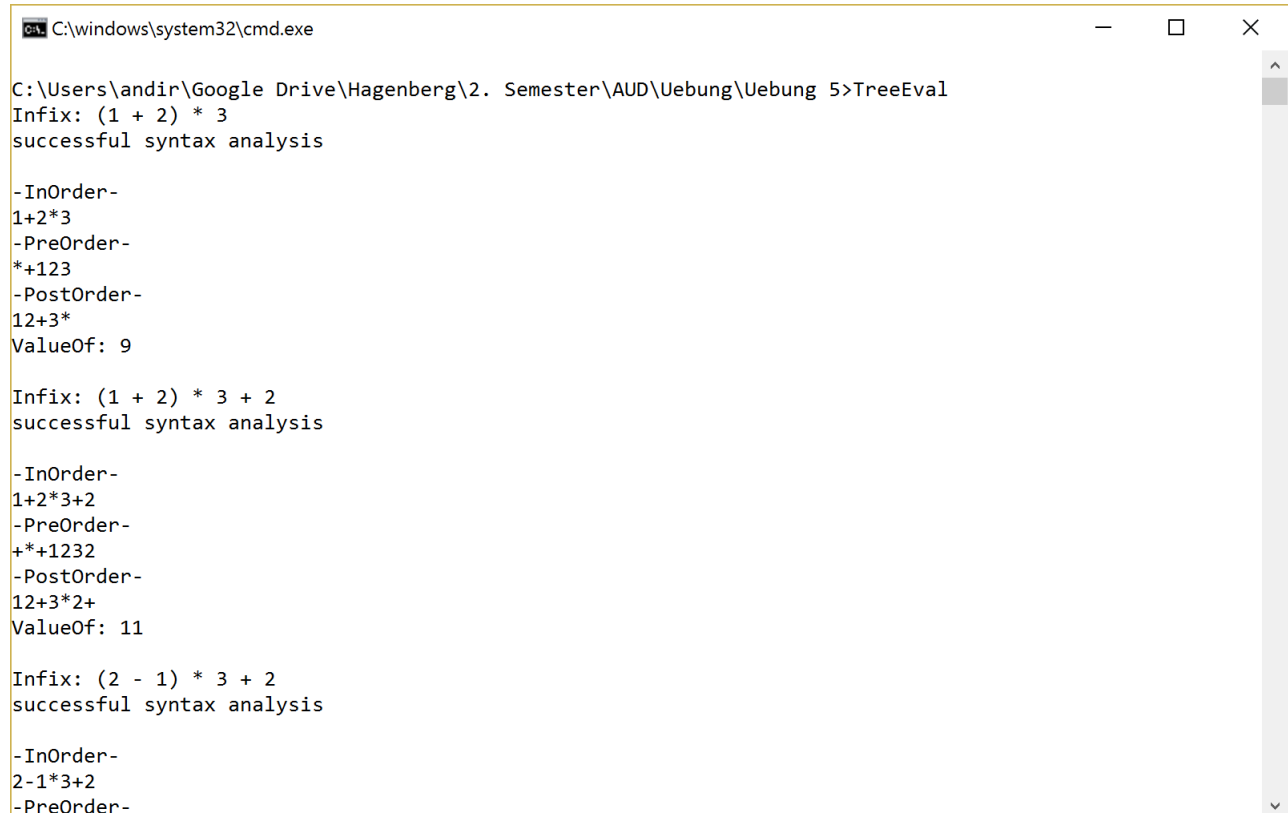
287  END;
288  END;
289
290  PROCEDURE DisposeTree(VAR t: TreePtr);
291  BEGIN
292      IF t <> NIL THEN BEGIN
293          DisposeTree(t^.left);
294          DisposeTree(t^.right);
295          Dispose(t);
296          t := NIL;
297      END;
298  END;
299
300  (* ===== END TREE ===== *)
301
302  PROCEDURE SyntaxTest(str: STRING);
303  BEGIN
304      WriteLn('Infix: ', str);
305      line := str;
306      cnr := 0;
307      NewCh;
308      NewSy;
309      success := TRUE;
310
311      S;
312      IF success THEN WriteLn('successful syntax analysis',#13#10) ELSE WriteLn('Error in
313          column: ', cnr,#13#10);
314
315      WriteLn('–InOrder–');
316      WriteTreeInOrder(tr);
317      WriteLn(#13#10, '–PreOrder–');
318      WriteTreePreOrder(tr);
319      WriteLn(#13#10, '–PostOrder–');
320      WriteTreePostOrder(tr);
321      WriteLn(#13#10, 'ValueOf: ', ValueOf(tr), #13#10);
322  END;
323
324  BEGIN
325      InitTree(tr);
326      SyntaxTest('(1 + 2) * 3');
327      DisposeTree(tr);
328
329      InitTree(tr);
330      SyntaxTest('(1 + 2) * 3 + 2');
331      DisposeTree(tr);
332
333      InitTree(tr);
334      SyntaxTest('(2 - 1) * 3 + 2');

```



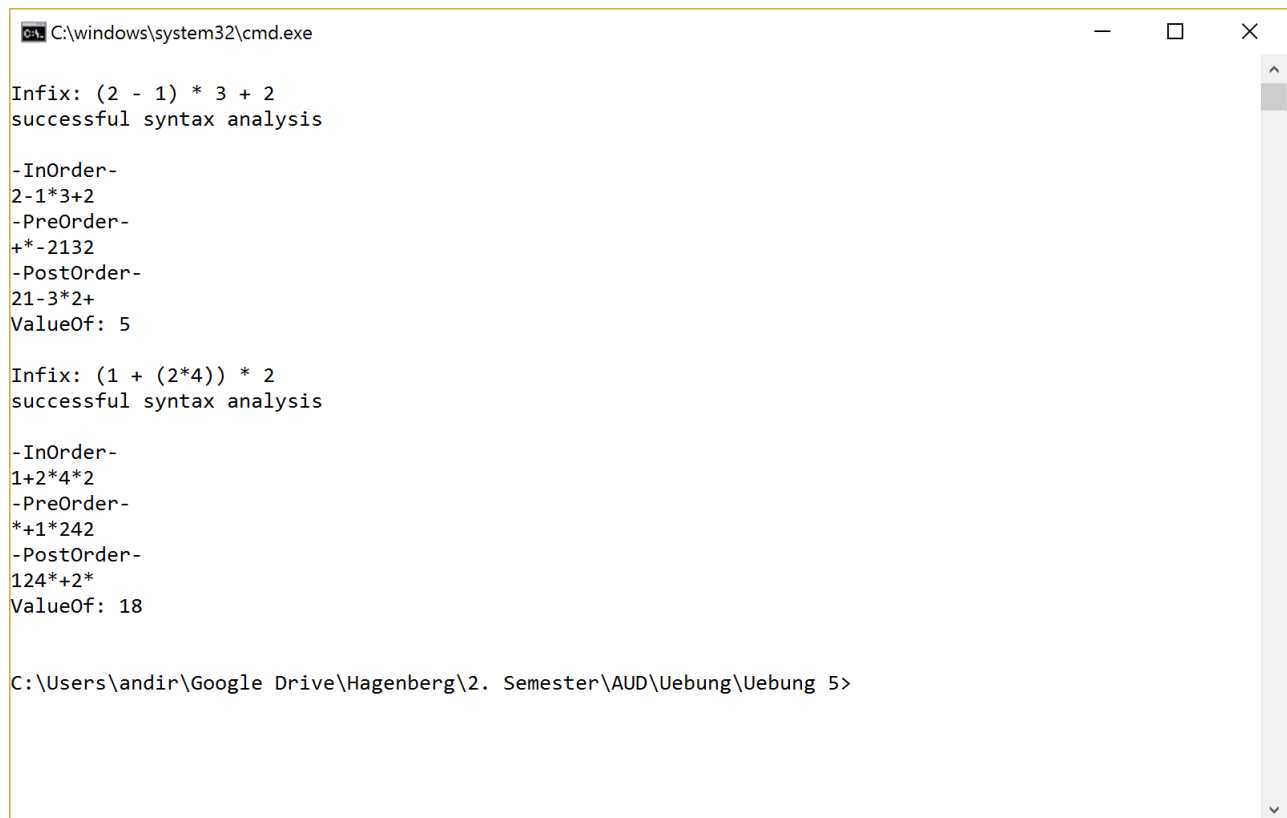
```
335 DisposeTree(tr);  
337 InitTree(tr);  
339 SyntaxTest('(1 + (2*4)) * 2');  
DisposeTree(tr);  
END.
```

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.