

<input checked="" type="checkbox"/> Gr. 1, Dr. G. Kronberger	Name <u>PAPESH Konstantin</u>	Aufwand in h <u>5</u>
<input type="checkbox"/> Gr. 2, Dr. H. Gruber		
<input type="checkbox"/> Gr. 3, Dr. D. Auer	Punkte <u>6,5</u>	Kurzzeichen Tutor / Übungsleiter <u>C.M.</u> / <u></u>

1. Syntaxbäume in kanonischer Form

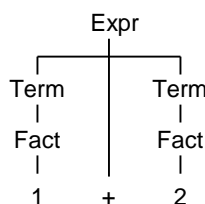
(10 Punkte)

Die Syntax einfacher arithmetischer Ausdrücke in Infix-Notation, z. B. $(17 + 4) * 21$, kann durch folgende Grammatik beschrieben werden:

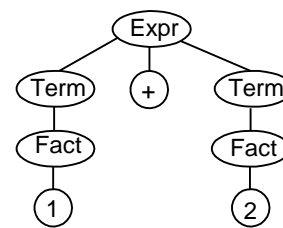
$\text{Expr} = \text{Term} \{ '+' \text{Term} \mid '-' \text{Term} \} .$
 $\text{Term} = \text{Fact} \{ '*' \text{Fact} \mid '/' \text{Fact} \} .$
 $\text{Fact} = \text{number} \mid '(' \text{Expr} ')'$

Die Struktur von solchen Ausdrücken kann auf Basis obiger Grammatik durch ihren Syntaxbaum dargestellt werden. Folgende Abbildungen zeigen zwei unterschiedliche Darstellungen des Syntaxbaums für den Beispielausdruck $1 + 2$:

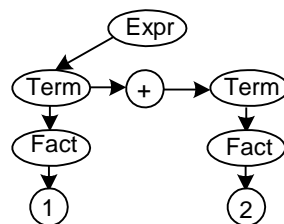
"Konventionelle" Darstellung



Baumdarstellung



Syntaxbäume sind somit Bäume, deren Knoten beliebig viele Söhne haben können. Will man Syntaxbäume in Form von dynamischen Datenstrukturen abbilden, tritt ein Problem auf: Wie viele Zeiger braucht ein Knoten? Eine einfache Implementierung für solche *allgemeinen Bäume* besteht darin, diese auf den Spezialfall der *Binärbäume* zurückzuführen, indem jeder Knoten einen Zeiger auf sein erstes Kind (in der Komponente *firstChild*) und einen Zeiger auf die einfach-verkettete Liste seiner Geschwister (in der Komponente *sibling*) hat. Jeder Knoten kommt also mit zwei Zeigern aus, und zwar unabhängig davon, wie viele Geschwister er hat. Man nennt diese Darstellung *kanonische Form*. Der Syntaxbaum für das obige Beispiel sieht in kanonischer Form wie folgt aus:



Entwickeln Sie aus der oben angegebenen Grammatik eine attributierte Grammatik (ATG), die für arithmetische Ausdrücke den Syntaxbaum in kanonischer Form erzeugt und implementieren Sie diese. Verwenden Sie dazu folgende Deklarationen:

```

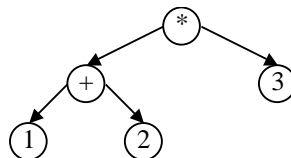
TYPE
  NodePtr = ^Node;
  Node = RECORD
    firstChild, sibling: NodePtr;
    val: STRING; (* nonterminal, operator or operand as text *)
  END; (*RECORD*)
  TreePtr = NodePtr;

```

2. Arithmetische Ausdrücke und Binärbäume

(4 + 6 + 2 + 2 Punkte)

Arithmetische Ausdrücke können auch in Form von („echten“) Binärbäumen dargestellt werden. Z. B. entspricht dem Ausdruck $(1 + 2) * 3$ der folgende Binärbaum:



Man nennt diese Darstellung auch *abstrakte Syntax*, weil es sich um eine abstraktere Darstellung (mit weniger Information) als die *konkrete Syntax* (gegeben durch den Syntaxbaum) handelt.

- a) Geben Sie eine attributierte Grammatik an, welche arithmetische Ausdrücke in Binärbäume (gemäß den unten angegebenen Deklarationen) umwandelt.

```

TYPE
  NodePtr = ^Node;
  Node = RECORD
    left, right: NodePtr;
    val: STRING; (* operator or operand as text *)
  END; (*RECORD*)
  TreePtr = NodePtr;

```

- b) Implementieren Sie diese attributierte Grammatik.
- c) Geben Sie die Ergebnisbäume durch entsprechende Baumdurchläufe *in-order*, *post-order* und *pre-order* aus: Was stellen Sie dabei (insbesondere auch durch Vergleich mit der ersten Aufgabe fest?
- d) 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. Dazu ist im Wesentlichen ein *post-order*-Baumdurchlauf erforderlich: Wert des linken Unterbaums berechnen, Wert des rechten Unterbaums berechnen und in Abhängigkeit vom Operator in der Wurzel den Gesamtwert berechnen.


ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP – SS 2018 Übungsabgabe 6

Konstantin Papesh

9. Mai 2018

6.1 Syntaxbäume in kanonischer Form

6.1.1 Lösungsidee

Mithilfe von attributierter Grammatik kann das Problem eines minimalistischen Taschenrechners relativ leicht behandelt werden. Dabei wird das Programm in verschiedene Module aufgebrochen, dem Syntaxparser *ModCalcSyn.pas* und dem Lexer *ModCalcLex.pas*. Im Lexer wird dann der Syntax definiert, wobei jedem Character ein Symbol zugemappt wird. Der Syntaxparser kümmert sich dann aufgrund der attribuierten Grammatik um die richtige Abwandlung der Eingabe. 

6.1.2 Implementierung

Listing 6.1: ModCalcLex.pas

```
1 unit ModCalcLex;
2
3 interface
4 type
5     symbol = (numberSy,
6               plusSy, minusSy,
7               mulSy, divSy,
8               leftParSy, rightParSy,
9               eofSy, noSy);
10 var
11     curSy : symbol;
12     numVal : integer; (* number value for semantic analysis *)
13
14 procedure newSy;
15 procedure initCalcLex(inFileName : string);
16
17 implementation
18 const EOF_CH = chr(26);
19       TAB_CH = chr(9);
20 var
21     inFile : text;
```

```

22   line : string;
23   curChPos : integer;
24   curCh : char;
25
26 procedure newCh; forward;
27
28 procedure initCalcLex(inFileName : string);
29 begin
30   assign(inFile, inFileName);
31   reset(inFile);
32   readLn(inFile, line);
33   curChPos := 0;
34   NewCh;
35 end;
36
37 procedure newSy;
38 begin
39   (* skip whitespace *)
40   while (curCh = ' ') or (curCh = TAB_CH) do newCh;
41   case curCh of
42     '+':   begin curSy := plusSy;   newCh; end;
43     '-':   begin curSy := minusSy;  newCh; end;
44     '*':   begin curSy := mulSy;    newCh; end;
45     '/':   begin curSy := divSy;    newCh; end;
46     '(':   begin curSy := leftParSy; newCh; end;
47     ')':   begin curSy := rightParSy; newCh; end;
48     EOF_CH: begin curSy := eofSy;    newCh; end;
49     '0'..'9': begin
50       (* read a number *)
51       numVal := Ord(curCh) - Ord('0'); (* value of digit*)
52       newCh;
53       while (curCh > '0') and (curCh <= '9') do begin
54         numVal := numVal * 10 + Ord(curCh) - Ord('0');
55         newCh;
56       end;
57       curSy := numberSy;
58     end;
59     else begin curSy := noSy; newCh; end; (* default case *)
60   end; (* case *)
61 end;
62
63 procedure newCh;
64 begin
65   if curChPos < length(line) then begin
66     inc(curChPos);
67     curCh := line[curChPos]
68   end else
69     curCh := EOF_CH;
70 end;
71
72 begin
73 end.

```

Listing 6.2: ModCalcSyn.pas

```

1 unit ModCalcSyn;
2

```

```

3 interface
4 var
5     success : boolean;
6 procedure S;
7
8 implementation
9 uses ModCalcLex, sysutils;
10
11 type
12     nodePtr = ^Node;
13     node = record
14         firstChild, sibling : nodePtr;
15         val : string; (* nonterminal, operator or operand as text *)
16     end;
17     treePtr = nodePtr;
18
19 procedure expr(var ePtr : treePtr); forward;
20 procedure term(var tPtr : treePtr); forward;
21 procedure fact(var fPtr : treePtr); forward;
22
23 procedure S;
24 var exprPtr : treePtr;
25 begin
26     new(exprPtr);
27     (* S = expr EOF. *)
28     success := TRUE;
29     expr(exprPtr); if not success then exit;
30     if curSy <> eofSy then begin success := FALSE; exit; end;
31     newSy;
32     (* sem *)
33     write(exprPtr^.val);
34     (* endsem *)
35 end;
36
37 procedure expr(var ePtr : treePtr);
38 var firstChildPtr : treePtr;
39     siblingPtr : treePtr;
40 begin
41     new(firstChildPtr);
42     new(siblingPtr);
43     (* Expr = Term { '+' Term | '-' Term } *)
44     term(firstChildPtr); if not success then exit;
45     while (curSy = plusSy) or (curSy = minusSy) do begin
46         case curSy of
47             plusSy: begin
48                 newSy;
49                 term(siblingPtr); if not success then exit;
50                 (* sem *)
51                 ePtr^.val := '+';
52                 ePtr^.firstChild := firstChildPtr;
53                 ePtr^.sibling := siblingPtr;
54                 (* endsem *)
55             end;
56             minusSy: begin
57                 newSy;
58                 term(siblingPtr); if not success then exit;
59                 (* sem *)

```

```

60         ePtr^.val := '-';
61         ePtr^.firstChild := firstChildPtr;
62         ePtr^.sibling := siblingPtr;
63         (* endsem *)
64     end;
65 end; (* case *)
66 end; (* while *)
67 end;
68
69 procedure term(var tPtr : treePtr);
70 var firstChildPtr : treePtr;
71     siblingPtr : treePtr;
72 begin
73     new(firstChildPtr);
74     new(siblingPtr);
75     (* Term = Fact { '*' Fact | '/' Fact }. *)
76     fact(firstChildPtr); if not success then exit;
77     while (curSy = mulSy) or (curSy = divSy) do begin
78         case curSy of
79             mulSy: begin
80                 newSy;
81                 fact(siblingPtr); if not success then exit;
82                 (* sem *)
83                 tPtr^.val := '*';
84                 tPtr^.firstChild := firstChildPtr;
85                 tPtr^.sibling := siblingPtr;
86                 (* endsem *)
87             end;
88             divSy: begin
89                 newSy;
90                 fact(siblingPtr); if not success then exit;
91                 (* sem *)
92                 tPtr^.val := '/';
93                 tPtr^.firstChild := firstChildPtr;
94                 tPtr^.sibling := siblingPtr;
95                 (* endsem *)
96             end;
97         end; (* case *)
98     end; (* while *)
99 end;
100
101 procedure fact(var fPtr : treePtr);
102 begin
103     (* Fact = number | '(' Expr ')' . *)
104     case curSy of
105         numberSy : begin
106             newSy;
107             (* sem *)
108             fPtr^.val := intToStr(numVal);
109             (* endsem *)
110         end;
111         leftParSy : begin
112             newSy; (* skip *)
113             expr(fPtr); if not success then exit;
114             if curSy <> rightParSy then begin success := FALSE; exit; end;
115             newSy;
116         end;

```

```

117         else begin
118             success := FALSE; exit;
119         end; (* else *)
120     end; (* case *)
121 end;
122
123 begin
124 end.

```

Listing 6.3: TestCalc.pas

```

1 program TestCalc;
2 uses ModCalcLex, ModCalcSyn;
3
4 var
5     inputFileName : string;
6
7 begin
8     inputFileName := '';
9     if ParamCount > 0 then
10         inputFileName := ParamStr(1);
11         initCalcLex(inputFileName);
12         newSy;
13         S; // read a sentence using procedure for sentence symbol S
14         writeln('Success: ', success);
15 end.

```



6.2 Arithmetische Ausdrücke und Binärbäume

6.2.1 Lösungsidee

Grundsätzlich entspricht die Implementierung der von 6.1. Nur kommt ein abstrakter Syntaxbaum zum Einsatz, sodass sich Knoten erspart werden. Der Lexer wird dabei nicht verändert.



6.2.2 Implementierung

Listing 6.4: ModCalcSynAbs.pas

```

1 unit ModCalcSynAbs;
2
3 interface
4 var
5     success : boolean;
6 procedure S;
7
8 implementation
9 uses ModCalcLex, sysutils;
10
11 type
12     nodePtr = ^Node;
13     node = record
14         left, right : nodePtr;
15         val : string; (* nonterminal, operator or operand as text *)
16     end;

```

```

17     treePtr = nodePtr;
18
19 procedure expr(var ePtr : treePtr); forward;
20 procedure term(var tPtr : treePtr); forward;
21 procedure fact(var fPtr : treePtr); forward;
22
23 procedure S;
24 var exprPtr : treePtr;
25 begin
26     new(exprPtr);
27     (* S = expr EOF. *)
28     success := TRUE;
29     expr(exprPtr); if not success then exit;
30     if curSy <> eofSy then begin success := FALSE; exit; end;
31     newSy;
32     (* sem *)
33     write(exprPtr^.val);
34     (* endsem *)
35 end;
36
37 procedure expr(var ePtr : treePtr);
38 var leftPtr : treePtr;
39     rightPtr : treePtr;
40 begin
41     new(leftPtr);
42     new(rightPtr);
43     (* Expr = Term { '+' Term | '-' Term } *)
44     term(leftPtr); if not success then exit;
45     while (curSy = plusSy) or (curSy = minusSy) do begin
46         case curSy of
47             plusSy: begin
48                 newSy;
49                 term(rightPtr); if not success then exit;
50                 (* sem *)
51                 ePtr^.val := '+';
52                 ePtr^.left := leftPtr;
53                 ePtr^.right := rightPtr;
54                 (* endsem *)
55             end;
56             minusSy: begin
57                 newSy;
58                 term(siblingPtr); if not success then exit;
59                 (* sem *)
60                 ePtr^.val := '-';
61                 ePtr^.left := leftPtr;
62                 ePtr^.right := siblingPtr;
63                 (* endsem *)
64             end;
65         end; (* case *)
66     end; (* while *)
67 end;
68
69 procedure term(var tPtr : treePtr);
70 var leftPtr : treePtr;
71     rightPtr : treePtr;
72 begin
73     new(leftPtr);

```



```

74   new(rightPtr);
75   (* Term = Fact { '*' Fact | '/' Fact }. *)
76   fact(leftPtr); if not success then exit;
77   while (curSy = mulSy) or (curSy = divSy) do begin
78       case curSy of
79           mulSy: begin
80               newSy;
81               fact(rightPtr); if not success then exit;
82               (* sem *)
83               tPtr^.val := '*';
84               tPtr^.left := leftPtr;
85               tPtr^.right := rightPtr;
86               (* endsem *)
87           end;
88           divSy: begin
89               newSy;
90               fact(rightPtr); if not success then exit;
91               (* sem *)
92               tPtr^.val := '/';
93               tPtr^.left := leftPtr;
94               tPtr^.right := rightPtr;
95               (* endsem *)
96           end;
97       end; (* case *)
98   end; (* while *)
99 end;
100
101 procedure fact(var fPtr : treePtr);
102 begin
103     (* Fact = number | '(' Expr ')' . *)
104     case curSy of
105         numberSy : begin
106             newSy;
107             (* sem *)
108             fPtr^.val := intToStr(numVal);
109             (* endsem *)
110         end;
111         leftParSy : begin
112             newSy; (* skip *)
113             expr(fPtr); if not success then exit;
114             if curSy <> rightParSy then begin success := FALSE; exit; end;
115             newSy;
116         end;
117         else begin
118             success := FALSE; exit;
119         end; (* else *)
120     end; (* case *)
121 end;
122
123 begin
124 end.

```

