Abgabetermin: Sa, 25.05.2024

☒ **Gr. 1,** S. Schöberl, MSc **Name** Elias Leonhardsberger **Aufwand in h** 10

☐ **Gr. 2,** DI (FH) G. Horn-Völlenkle, MSc

Punkte _____ **Tutor*in / Übungsleiter*in** ____ / ____

1. ATG für Eishockey-Spielbericht (6 Punkte)

Für die Eishockey-Weltmeisterschaft der Herren 2024 wird ein Softwaresystem zur Auswertung von Spielberichten benötigt. Jeder Spielbericht enthält für jeden Spieler eine Liste von Ereignissen, an denen er beteiligt ist. Es gibt zwei Arten von Ereignissen: Ein Spieler erzielt ein Tor (`TOR`) oder er erhält eine Strafe (`STRAFE`). Für jedes Ereignis wird die Ereignisart und die Zeit in Form `MM:SS` aufgezeichnet; für Strafen zusätzlich die Strafzeit in Minuten.

Beispiel:

```
Vanek (TOR 04:45, STRAFE 17:34 2, TOR 34:34)
Lukas (STRAFE 45:23 2)
Steward (STRAFE 03:45 2, STRAFE 22:55 5, TOR 34:12, STRAFE 57:06 2)
Peintner ()
```

Für einen Spielbericht müssen die Anzahl der Tore und die Summe der Strafminuten pro Spieler ausgegeben werden. Zusätzlich ist die Summe der Strafminuten aller Spieler auszugeben.

Beispiel (Ausgabe):

```
Vanek: Tore: 2, Strafminuten: 2
Lukas: Tore: 0, Strafminuten: 2
Steward: Tore: 1, Strafminuten: 9
Peintner: Tore: 0, Strafminuten: 0
Gesamt: 13 Strafminuten
```

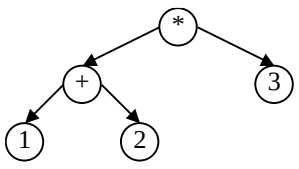
Entwickeln Sie als Grundlage für ein Softwaresystem, das diese Aufgabe löst, eine *attributierte Grammatik*. Für diese Aufgabe ist kein Pascal-Programm zu entwickeln!

2. Arithmetische Ausdrücke und Binärbäume (4 + 6 + 2 + 3 + 3 Punkte)

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

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

Die Struktur von solchen Ausdrücken kann durch einen Binärbaum dargestellt werden. Z. B. entspricht dem Ausdruck (1 + 2) * 3 der folgende Binärbaum:



Man nennt diese Darstellung auch *abstrakte Syntax* und den Binärbaum *abstrakten Syntaxbaum* (engl. *abstract syntax tree*, kurz *AST*) weil es sich um eine abstraktere Darstellung (mit weniger Information) als die *konkrete Syntax* (gegeben durch einen Parse-Baum, siehe Vorlesung) 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 string *)
  END; (* Node *)
  TreePtr = NodePtr;
```

b) Implementieren Sie diese attributierte Grammatik.

c) Geben Sie die Ergebnisbäume durch entsprechende Baumdurchläufe *in-order*, *post-order* und *pre-order* aus.

d) Für Testzwecke möchten Sie nun eine grafische Ausgabe der abstrakten Syntaxbäume erzeugen, indem Sie *Graphviz* einsetzen (https://graphviz.org). Das Werkzeug *Graphviz* erzeugt aus einer textuellen Beschreibung der Knoten und Kanten eines Graphen (oder eines Baumes) eine Grafik. Zum Beispiel ergibt die folgende Beschreibung die Baumdarstellung für den Ausdruck (1 + 2) * 3.

```
digraph G {
  n5 [label="*"];
  n3 [label="+"];
  n1 [label="1"];
  n2 [label="2"];
  n4 [label="3"];
  n5 -> n3;
  n5 -> n4;
  n3 -> n1;
  n3 -> n2;
}
```

Erweitern Sie den Datentyp `Node` um eine eindeutige ID für Knoten und implementieren Sie Prozeduren, um aus einem abstrakten Syntaxbaum eine passende Graphviz-Beschreibung zu erzeugen. Testen Sie Ihre Implementierung mit verschiedenen arithmetischen Ausdrücken und geben Sie mit Ihrer Lösung die generierten Beschreibungen und die von Graphviz erzeugten Grafiken ab.

Das Werkzeug Graphviz können Sie lokal installieren (https://graphviz.org/) oder online verwenden, z.B. hier: https://dreampuf.github.io/GraphvizOnline.

e) Implementieren Sie eine rekursive Funktion

```
FUNCTION ValueOf(t: TreePtr): INTEGER;
```

die 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.

Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine „Lösungsidee“ an.
2. Dokumentieren und kommentieren Sie Ihre Algorithmen.
3. Bei Programmen: Geben Sie immer auch Testfälle ab, an denen man erkennen kann, dass Ihr Programm funktioniert, und dass es auch in Fehlersituation entsprechend reagiert.

ADF2/PRO2 UE05

Elias Leonhardsberger

24. Mai 2024, Hagenberg

Inhaltsverzeichnis

1 ATG für Eishockey-Spielbericht

1.1 Lösungsidee

Für die EBNF wird das Latex package "naive-ebnf"verwendet.

Für die erste Angabe wird angenommen das die Ausgabe direkt mit *Write()* ausgegeben wird, also nicht als *STRING* zurückgegeben.

Jede Zeile besteht aus einem Namen(Name), zwei Klammern(leftPar, rightPar) und den Ereignissen(Event) dazwischen.

Ein Ereignis ist entweder ein Tor oder eine Strafe, gefolgt von einer Uhrzeit(time) und bei der Strafe eine Minutenanzahl(number).

Nach jeder Zeile wird diese ausgegeben und eine Summe an Strafminuten wird weitergegeben um diese am Schluss auszugeben.

Es wird angenommen, dass die Großkleinschreibung bei Namen egal ist und keine Sonderzeichen erlaubt sind.

Die Minutenanzahl ist durch Overtime nur an zweistellige Werte begrenzt.

1.2 Attributierte Gramatik

1.2.1 Keywords

"TOR" .
"STRAFE" .

1.2.2 Terminalsymbole

⟨**leftPar**⟩ → "(" .
⟨**rightPar**⟩ → ")" .
⟨**comma**⟩ → "," .

1.2.3 Terminalklassen

⟨**name**$_{\uparrow \textbf{name}}$⟩ → /[A-Z,a-z]+/ .

⟨**digit**$_{\uparrow \textbf{value}}$⟩ → /[0-9]/ .

⟨**number**$_{\uparrow \textbf{value}}$⟩ →
⟨digit$_{\uparrow \text{digitValue}}$⟩
{⟨digit$_{\uparrow \text{digitValue}}$⟩} .

⟨local⟩VAR digitValue: INTEGER;⟨endlocal⟩
⟨sem⟩value → digitValue;⟨endsem⟩
⟨sem⟩value → value * 10 + digitValue;⟨endsem⟩

⟨**time**⟩ → ⟨digit⟩ ⟨digit⟩ ":" ⟨digit⟩ ⟨digit⟩ .

1.2.4 Nonterminalsymbole

⟨**Report**⟩ →

⟨local⟩
  VAR penalties, playerPenalties: INTEGER;
⟨endlocal⟩
⟨sem⟩penalties → 0;⟨endsem⟩

{⟨Player$_{\uparrow \text{playerPenalties}}$⟩}

⟨sem⟩
  penalties → penalties + playerPenalties;
⟨endsem⟩

.

⟨sem⟩
  WriteLn("`Gesamt: `", penalties, " `Strafminuten`");
⟨endsem⟩

⟨**Player$_{\uparrow \text{penalties}}$**⟩ →

⟨local⟩
  VAR
    goals, g, p: INTEGER;
    name: STRING;
⟨endlocal⟩

⟨name$_{\uparrow \text{name}}$⟩

⟨sem⟩
  Write(name, "`: `");
  goals → 0;
  penalties → 0;
⟨endsem⟩

⟨leftPar⟩ [⟨Event$_{\uparrow g \uparrow p}$⟩

⟨sem⟩
  penalties → p;
  goals → g;
⟨endsem⟩

{⟨comma⟩ ⟨Event$_{\uparrow g \uparrow p}$⟩}

⟨sem⟩
  penalties → penalties + p;
  goals → goals + g;
⟨endsem⟩

] ⟨rightpar⟩ .

⟨sem⟩
  Write("`Tore: `", goals);
  WriteLn("`, Strafminuten`", penalties);
⟨endsem⟩

⟨**Event$_{\uparrow g \uparrow p}$**⟩ →

⟨local⟩VAR temp: INTEGER;⟨endlocal⟩

("`TOR`" ⟨time⟩

⟨sem⟩
  goals → 1;
  penalties → 0;
⟨endsem⟩

| "`STRAFE`" ⟨time⟩ ⟨number⟩

⟨sem⟩
  g → 0;
  p → temp;
⟨endsem⟩

) .

2 Arithmetische Ausdrücke und Binärbäume

2.1 Lösungsidee

Für den Binärbaum wird die bereits vorhandene Grammatik und der Code der Übung übernommen und angepasst.

Der Baum wird in einem eigenen Unit implementiert um die verschiedenen Ausgabearten unabhängig zu implementieren. Der Datentyp wird nicht besonders versteckt.

Die Ausgabe wird rekursiv in der angegebenen Reihenfolge implementiert.

Die Id für den Graphen wird direkt im Parser erstellt. Der Graph selbst wird rekursiv mit einer definierten Eingangsfunktion erstellt, wobei zuerst die Labels und danach die Edges erstellt werden. Der Graph wird direkt ausgegeben.

Die Tests sind in einem shell Script geschrieben und mit folgendem Befehl ausgeführt worden.

```
./TestScript.sh &> ./result.txt
```

2.2 Attributierte Gramatik

2.2.1 Terminalsymbole

⟨**leftPar**⟩ → "(" .
⟨**rightPar**⟩ → ")" .
⟨**plus**⟩ → "+" .
⟨**minus**⟩ → "-" .
⟨**mult**⟩ → "*" .
⟨**div**⟩ → "/" .

2.2.2 Terminalklassen

⟨**number**$_{\uparrow \textbf{stringVal}}$⟩ → /[0-9]+/

2.2.3 Nonterminalsymbole

⟨**Expr**$_{\uparrow \textbf{e}}$⟩ →

⟨Term$_{\uparrow t}$⟩
{(⟨plus⟩ ⟨Term$_{\uparrow t}$⟩
| ⟨minus⟩ ⟨Term$_{\uparrow t}$⟩
)

} .

⟨local⟩
  VAR
    temp, t: NodePtr;
    operator: CHAR;
⟨endlocal⟩
⟨sem⟩e → t;⟨endsem⟩
⟨sem⟩operator → "+";⟨endsem⟩
⟨sem⟩operator → "-";⟨endsem⟩
⟨sem⟩
  temp → new(NodePtr);
  temp^.val → operator;
  temp^.left → e;
  temp^.right → t;
  e → temp;
⟨endsem⟩

⟨**Term**$_{\uparrow \mathbf{t}}$⟩ →

⟨local⟩
VAR
temp, f: NodePtr;
operator: CHAR;
⟨endlocal⟩

⟨Fact$_{\uparrow \mathsf{f}}$⟩
(⟨mult⟩ ⟨Fact$_{\uparrow \mathsf{f}}$⟩
| ⟨div⟩ ⟨Fact$_{\uparrow \mathsf{f}}$⟩
)

⟨sem⟩left → f;⟨endsem⟩
⟨sem⟩operator → "*";⟨endsem⟩
⟨sem⟩operator → "/";⟨endsem⟩
⟨sem⟩
temp → new(NodePtr);
tempˆ.val → operator;
tempˆ.left → t;
tempˆ.right → f;
t → temp;
⟨endsem⟩

.

⟨**Fact**$_{\uparrow \mathbf{f}}$⟩ →

⟨local⟩
VAR
e: NodePtr;
n: STRING;
⟨endlocal⟩

(⟨number$_{\uparrow \mathsf{n}}$⟩

⟨sem⟩
f → new(NodePtr);
fˆ.val → s;
fˆ.left → NIL;
fˆ.right → NIL;
⟨endsem⟩

| ⟨leftPar⟩ ⟨Expr$_{\uparrow \mathsf{e}}$⟩
⟨rightPar⟩) .

⟨sem⟩f → e⟨endsem⟩

2.3 Souce Code

2.3.1 ExpressionTree.pas

```
1
2  UNIT ExpressionTree;
3
4  INTERFACE
5
6  TYPE
7    NodePtr = ^Node;
8    Node = RECORD
9      left, right: NodePtr;
10     val, id: STRING;
11   END;
12   TreePtr = NodePtr;
13
14 PROCEDURE DisposeTree(VAR root: TreePtr);
15
16 PROCEDURE PrintTreeInOrder(root: TreePtr);
17
18 PROCEDURE PrintTreePostOrder(root: TreePtr);
19
20 PROCEDURE PrintTreePreOrder(root: TreePtr);
21
22 FUNCTION ValueOf(root: TreePtr): INTEGER;
23
24 PROCEDURE PrintGraphviz(root: TreePtr);
25
26 IMPLEMENTATION
27
28 PROCEDURE DisposeTree(VAR root: TreePtr);
29 BEGIN (* DisposeTree *)
30   IF (root <> NIL) THEN
31     BEGIN (* IF *)
32       DisposeTree(root^.left);
33       DisposeTree(root^.right);
34       DISPOSE(root);
35       root := NIL;
36     END; (* IF *)
37 END; (* DisposeTree *)
38
39 PROCEDURE PrintTreeInOrder(root: TreePtr);
40 BEGIN (* PrintTreeInOrder *)
41   IF (root <> NIL) THEN
42     BEGIN (* IF *)
43       PrintTreeInOrder(root^.left);
44       Write(root^.val, ' ');
```

```
45        PrintTreeInOrder(root^.right);
46      END; (* IF *)
47  END; (* PrintTreeInOrder *)
48
49  PROCEDURE PrintTreePostOrder(root: TreePtr);
50  BEGIN (* PrintTreePostOrder *)
51    IF (root <> NIL) THEN
52      BEGIN (* IF *)
53        PrintTreePostOrder(root^.left);
54        PrintTreePostOrder(root^.right);
55        Write(root^.val, ' ');
56      END; (* IF *)
57  END; (* PrintTreePostOrder *)
58
59  PROCEDURE PrintTreePreOrder(root: TreePtr);
60  BEGIN (* PrintTreePreOrder *)
61    IF (root <> NIL) THEN
62      BEGIN (* IF *)
63        Write(root^.val, ' ');
64        PrintTreePreOrder(root^.left);
65        PrintTreePreOrder(root^.right);
66      END; (* IF *)
67  END; (* PrintTreePreOrder *)
68
69  FUNCTION ValueOf(root: TreePtr): INTEGER;
70  VAR
71    temp, code: INTEGER;
72  BEGIN (* ValueOf *)
73    IF (root = NIL) THEN
74      BEGIN (* IF *)
75        WriteLn('Error: Empty (Sub)Tree');
76        HALT(1);
77      END; (* IF *)
78
79    IF (root^.val = '+') THEN
80      BEGIN (* IF *)
81        ValueOf := ValueOf(root^.left) + ValueOf(root^.right)
82      END (* IF *)
83    ELSE
84      IF (root^.val = '-') THEN
85        BEGIN (* ELSE IF *)
86          ValueOf := ValueOf(root^.left) - ValueOf(root^.right)
87        END (* ELSE IF *)
88    ELSE
89      IF (root^.val = '*') THEN
90        BEGIN (* ELSE IF *)
91          ValueOf := ValueOf(root^.left) * ValueOf(root^.right)
```

```
92        END (* ELSE IF *)
93    ELSE
94      IF (root^.val = '/') THEN
95        BEGIN (* ELSE IF *)
96          temp := ValueOf(root^.right);
97
98          IF (temp = 0) THEN
99            BEGIN (* IF *)
100             WriteLn('Error: Division by Zero');
101             HALT(1);
102           END; (* IF *)
103
104         ValueOf := ValueOf(root^.left) DIV temp;
105       END (* ELSE IF *)
106   ELSE
107     BEGIN (* ELSE *)
108       Val(root^.val, temp, code);
109
110       IF (code <> 0) THEN
111         BEGIN (* IF *)
112           WriteLn('Error: Invalid Number, Error Code: ', code);
113           HALT(1);
114         END; (* IF *)
115
116       ValueOf := temp;
117     END; (* ELSE *)
118 END; (* ValueOf *)
119
120 PROCEDURE PrintGraphvizNode(root: NodePtr);
121 BEGIN (* PrintGraphvizNode *)
122   WriteLn(root^.id + ' [label="' + root^.val + '"];');
123
124   IF (root^.left <> NIL) THEN
125     BEGIN (* IF *)
126       PrintGraphvizNode(root^.left);
127       WriteLn(root^.id + ' -> ' + root^.left^.id + '; ');
128     END; (* IF *)
129
130   IF (root^.right <> NIL) THEN
131     BEGIN (* IF *)
132       PrintGraphvizNode(root^.right);
133       WriteLn(root^.id + ' -> ' + root^.right^.id + '; ');
134     END; (* IF *)
135 END; (* PrintGraphvizNode *)
136
137 PROCEDURE PrintGraphviz(root: TreePtr);
138 BEGIN (* PrintGraphviz *)
```

```
  WriteLn('digraph G {');
  PrintGraphvizNode(root);
  WriteLn('}');
END; (* PrintGraphviz *)

END.
```

2.3.2 ExpressionScanner.pas

```

UNIT ExpressionScanner;

INTERFACE

TYPE
  Symbol = (
             noSym,
             plusSym, minusSym, multSym, divSym,
             leftParSym, rightParSym,
             numberSym
            );

PROCEDURE InitScanner(VAR inputFile: Text);

PROCEDURE ReadNextSymbol;

FUNCTION GetCurrentSymbol: Symbol;

PROCEDURE GetCurrentSymbolPosition(VAR line, col: INTEGER);

FUNCTION GetCurrentNumber: STRING;

IMPLEMENTATION

CONST
  Tab = CHR(9);
  LF = CHR(10);
  CR = CHR(13);
  Space = ' ';

VAR
  currentSymbol: Symbol;
  currentLine, currentCol, symbolLine, symbolCol: INTEGER;
  currentNumber: STRING;
  currentChar: CHAR;
  inFile: Text;

PROCEDURE ReadNextChar;
```

```
40 BEGIN (* ReadNextChar *)
41   Read(inFile, currentChar);
42
43   IF (currentChar = LF) THEN
44     BEGIN (* IF *)
45       currentLine := currentLine + 1;
46       currentCol := 0;
47     END (* IF *)
48   ELSE
49     BEGIN (* ELSE *)
50       currentCol := currentCol + 1;
51     END; (* ELSE *)
52 END; (* ReadNextChar *)
53
54 PROCEDURE InitScanner(VAR inputFile: Text);
55 BEGIN (* InitScanner *)
56   inFile := inputFile;
57   currentLine := 1;
58   currentCol := 0;
59   ReadNextChar();
60   ReadNextSymbol();
61 END; (* InitScanner *)
62
63 PROCEDURE ReadNextSymbol;
64 BEGIN (* ReadNextSymbol *)
65   WHILE (currentChar = Space) OR (currentChar = LF) OR (currentChar = CR)
   ↪  OR (currentChar = Tab) DO
66     BEGIN (* WHILE *)
67       ReadNextChar();
68     END; (* WHILE *)
69
70   symbolLine := currentLine;
71   symbolCol := currentCol;
72
73   CASE currentChar OF
74     '+':
75          BEGIN (* + *)
76            currentSymbol := plusSym;
77            ReadNextChar();
78          END; (* + *)
79     '-':
80          BEGIN (* - *)
81            currentSymbol := minusSym;
82            ReadNextChar();
83          END; (* - *)
84     '*':
85          BEGIN (* * *)
```

```
86              currentSymbol := multSym;
87              ReadNextChar();
88            END; (* * *)
89      '/':
90            BEGIN (* / *)
91              currentSymbol := divSym;
92              ReadNextChar();
93            END; (* / *)
94      '(':
95            BEGIN (* ( *)
96              currentSymbol := leftParSym;
97              ReadNextChar();
98            END; (* ( *)
99      ')':
100           BEGIN (* ) *)
101             currentSymbol := rightParSym;
102             ReadNextChar();
103           END; (* ) *)
104     '0'..'9':
105               BEGIN (* number *)
106                 currentSymbol := numberSym;
107                 currentNumber := '';
108
109                 WHILE currentChar IN ['0' .. '9'] DO
110                   BEGIN (* WHILE *)
111                     currentNumber := currentNumber + currentChar;
112                     ReadNextChar();
113                   END; (* WHILE *)
114               END; (* number *)
115     ELSE
116       BEGIN (* ELSE *)
117         currentSymbol := noSym;
118       END; (* ELSE *)
119   END; (* CASE *)
120 END; (* ReadNextSymbol *)
121
122 FUNCTION GetCurrentSymbol: Symbol;
123 BEGIN (* GetCurrentSymbol *)
124   GetCurrentSymbol := currentSymbol;
125 END; (* GetCurrentSymbol *)
126
127 PROCEDURE GetCurrentSymbolPosition(VAR line, col: INTEGER);
128 BEGIN (* GetCurrentSymbolPosition *)
129   line := symbolLine;
130   col := symbolCol;
131 END; (* GetCurrentSymbolPosition *)
132
```

```
133 FUNCTION GetCurrentNumber: STRING;
134 BEGIN (* GetCurrentNumber *)
135   GetCurrentNumber := currentNumber;
136 END; (* GetCurrentNumber *)
137
138 END.
```

2.3.3 ExpressionParser.pas

```
1
2  UNIT ExpressionParser;
3
4  INTERFACE
5
6  USES
7  ExpressionTree;
8
9  PROCEDURE Parse(VAR inputFile: TEXT; VAR ok: BOOLEAN; VAR errorLine,
   ↪  errorCol: INTEGER; VAR result: TreePtr; VAR errorMessage: STRING);
10
11 IMPLEMENTATION
12
13 USES
14 ExpressionScanner;
15
16 VAR
17   success: BOOLEAN;
18   errMessage: STRING;
19   id: INTEGER;
20
21 PROCEDURE SemErr(message: STRING);
22 BEGIN (* SemErr *)
23   success := FALSE;
24   errMessage := message;
25 END; (* SemErr *)
26
27 PROCEDURE Expr(VAR e: NodePtr); FORWARD;
28 PROCEDURE Term(VAR t: NodePtr); FORWARD;
29 PROCEDURE Fact(VAR f: NodePtr); FORWARD;
30
31 PROCEDURE Parse(VAR inputFile: TEXT; VAR ok: BOOLEAN; VAR errorLine,
   ↪  errorCol: INTEGER; VAR result: TreePtr; VAR errorMessage: STRING);
32 BEGIN (* Parse *)
33   success := TRUE;
34   errMessage := '';
35   id := 1;
36
37   InitScanner(inputFile);
```

```
38
39    Expr(result);
40
41    GetCurrentSymbolPosition(errorLine, errorCol);
42    ok := success;
43    errorMessage := errMessage;
44  END; (* Parse *)
45
46  PROCEDURE Expr(VAR e: NodePtr);
47  {local}
48  VAR
49    temp, t: NodePtr;
50    operatorChar: CHAR;
51  {endlocal}
52  BEGIN (* Expr *)
53    Term(e);
54    IF NOT success THEN EXIT;
55
56    WHILE (GetCurrentSymbol() = plusSym) OR (GetCurrentSymbol() = minusSym)
      ↪  DO
57      BEGIN (* WHILE *)
58        CASE GetCurrentSymbol() OF
59          plusSym:
60                    BEGIN (* plusSym *)
61                      ReadNextSymbol();
62                      Term(t);
63                      {sem} operatorChar := '+'; {endsem}
64                      IF NOT success THEN EXIT;
65                    END; (* plusSym *)
66          minusSym:
67                     BEGIN (* minusSym *)
68                       ReadNextSymbol();
69                       Term(t);
70                       {sem} operatorChar := '-'; {endsem}
71                       IF NOT success THEN EXIT;
72                     END; (* minusSym *)
73        END; (* CASE *)
74
75        {sem}
76        temp := NEW(NodePtr);
77
78        IF (temp = NIL) THEN
79          BEGIN (* IF *)
80            SemErr('Out of memory');
81            EXIT;
82          END; (* IF *)
83
```

```
84        Str(id, temp^.id);
85        temp^.id := 'n' + temp^.id;
86        id := id + 1;
87        temp^.val := operatorChar;
88        temp^.left := e;
89        temp^.right := t;
90        e := temp;
91        {endsem}
92      END; (* WHILE *)
93  END; (* Expr *)
94
95  PROCEDURE Term(VAR t: NodePtr);
96  {local}
97  VAR
98    temp, f: NodePtr;
99    operatorChar: CHAR;
100 {endlocal}
101 BEGIN (* Term *)
102   Fact(t);
103   IF NOT success THEN EXIT;
104
105   WHILE (GetCurrentSymbol() = multSym) OR (GetCurrentSymbol() = divSym)
      ↪  DO
106     BEGIN (* WHILE *)
107       CASE GetCurrentSymbol() OF
108         multSym:
109                   BEGIN (* multSym *)
110                     ReadNextSymbol();
111                     Fact(f);
112                     {sem} operatorChar := '*'; {endsem}
113                     IF NOT success THEN EXIT;
114                   END; (* multSym *)
115         divSym:
116                  BEGIN (* divSym *)
117                    ReadNextSymbol();
118                    Fact(f);
119                     {sem} operatorChar := '/'; {endsem}
120                    IF NOT success THEN EXIT;
121                  END; (* divSym *)
122       END; (* CASE *)
123
124       {sem}
125       temp := NEW(NodePtr);
126
127       IF (temp = NIL) THEN
128         BEGIN (* IF *)
129           SemErr('Out of memory');
```

```
130             EXIT
131           END; (* IF *)
132
133         Str(id, temp^.id);
134         temp^.id := 'n' + temp^.id;
135         id := id + 1;
136         temp^.val := operatorChar;
137         temp^.left := t;
138         temp^.right := f;
139         t := temp;
140         {endsem}
141       END; (* WHILE *)
142   END; (* Term *)
143
144   PROCEDURE Fact(VAR f: NodePtr);
145   BEGIN (* Fact *)
146     CASE GetCurrentSymbol() OF
147       numberSym:
148                   BEGIN (* numberSym *)
149                     ReadNextSymbol();
150
151                    {sem}
152                     f := NEW(NodePtr);
153
154                     IF (f = NIL) THEN
155                       BEGIN (* IF *)
156                         SemErr('Out of memory');
157                         EXIT;
158                       END; (* IF *)
159
160                     Str(id, f^.id);
161                     f^.id := 'n' + f^.id;
162                     id := id + 1;
163                     f^.val := GetCurrentNumber();
164                     f^.left := NIL;
165                     f^.right := NIL;
166                    {endsem}
167                   END; (* numberSym *)
168       leftParSym:
169                    BEGIN (* leftParSym *)
170                      ReadNextSymbol();
171                      Expr(f);
172                      IF NOT success THEN EXIT;
173
174                      IF GetCurrentSymbol() <> rightParSym THEN
175                        BEGIN (* IF *)
176                          success := FALSE;
```

```
177                       EXIT;
178                     END; (* IF *)
179
180                   ReadNextSymbol();
181                 END; (* leftParSym *)
182     ELSE
183       BEGIN (* ELSE *)
184         success := FALSE;
185         EXIT;
186       END; (* ELSE *)
187   END; (* CASE *)
188 END; (* Fact *)
189
190 END.
```

2.3.4 ExprParse.pas

```
1  PROGRAM ExprParse;
2
3  USES
4  ExpressionParser, ExpressionTree;
5
6  VAR
7    f: TEXT;
8    errorCode: BYTE;
9    ok: BOOLEAN;
10   errorCol, errorLine: INTEGER;
11   tree: TreePtr;
12   errorMessage: STRING;
13 BEGIN (* ExprParse *)
14   IF (ParamCount <> 1) THEN
15     BEGIN (* IF *)
16       WriteLn('Usage: ExprParse <input file>');
17       Halt(1);
18     END; (* IF *)
19
20   Assign(f, ParamStr(1));
21
22   {$I-}
23   Reset(f);
24   {$I+}
25
26   errorCode := IOResult;
27
28   IF (errorCode <> 0) THEN
29     BEGIN (* IF *)
30       WriteLn('Error opening file: ', ParamStr(1), ' (error code: ',
       ↪  errorCode, ')');
```

```
31        Halt(1);
32      END; (* IF *)
33
34    Parse(f, ok, errorLine, errorCol, tree, errorMessage);
35    Close(f);
36
37    IF ok THEN
38      BEGIN (* IF *)
39        WriteLn('InOrder: ');
40        PrintTreeInOrder(tree);
41        WriteLn();
42        WriteLn('PostOrder: ');
43        PrintTreePostOrder(tree);
44        WriteLn();
45        WriteLn('PreOrder: ');
46        PrintTreePreOrder(tree);
47        WriteLn();
48        WriteLn('Graphviz: ');
49        PrintGraphviz(tree);
50        WriteLn();
51        WriteLn('Result: ', ValueOf(tree));
52        DisposeTree(tree);
53      END (* IF *)
54    ELSE
55      IF (errorMessage <> '') THEN
56        BEGIN (* ELSE IF *)
57          WriteLn('Error: ', errorMessage, ' at line ', errorLine, ' column
              ↪  ', errorCol);
58        END (* ELSE IF *)
59    ELSE
60      BEGIN (* ELSE *)
61        WriteLn('Syntax error at line ', errorLine, ' column ', errorCol);
62      END; (* ELSE *)
63  END. (* ExprParse *)
```

2.4 Tests

2.4.1 Testskript

```
echo "Invalid File Test":
../bin/ExprParse /invalidFile.txt
echo

echo "Syntax Error Test":
../bin/ExprParse syntaxError.txt
echo

echo "Division by Zero Test":
../bin/ExprParse divisionByZero.txt
echo

echo "Easy Expression Test":
../bin/ExprParse easyExpression.txt
echo

echo "Difficult Expression Test":
../bin/ExprParse difficultExpression.txt
echo
```

2.4.2 syntaxError.txt

Listing 1: syntaxError.txt

```
(1 + 2  3)
```

2.4.3 divisionByZero.txt

Listing 2: divisionByZero.txt

```
(4 + 5 + 2)/ (1 * 3 * 15 - 5 * 9)
```

2.4.4 easyExpression.txt

Listing 3: easyExpression.txt

```
 (1 + 2) * 3
```

2.4.5 difficultExpression.txt

Listing 4: difficultExpression.txt

```
(50 / 2 * ((25 - 14) * (1 + 14)
+ 14 * (1 - 25 + 14)))
+ (20 + 2 * 22 - 23)
```

2.5 Testergebnisse

2.5.1 Ausgabe des Testskripts

Listing 5: result.txt

```
Invalid File Test:
Error opening file: /invalidFile.txt (error code: 2)

Syntax Error Test:
Syntax error at line 1 column 8

Division by Zero Test:
InOrder:
4 + 5 + 2 / 1 * 3 * 15 - 5 * 9
PostOrder:
4 5 + 2 + 1 3 * 15 * 5 9 * - /
PreOrder:
/ + + 4 5 2 - * * 1 3 15 * 5 9
Graphviz:
digraph G {
n15 [label="/"];
n5 [label="+"];
n3 [label="+"];
n1 [label="4"];
n3 -> n1;
n2 [label="5"];
n3 -> n2;
n5 -> n3;
n4 [label="2"];
n5 -> n4;
n15 -> n5;
n14 [label="-"];
n10 [label="*"];
n8 [label="*"];
n6 [label="1"];
n8 -> n6;
n7 [label="3"];
n8 -> n7;
n10 -> n8;
n9 [label="15"];
n10 -> n9;
n14 -> n10;
n13 [label="*"];
n11 [label="5"];
n13 -> n11;
n12 [label="9"];
n13 -> n12;
```

```
n14 -> n13;
n15 -> n14;
}
```

```
Result: Error: Division by Zero
```

```
Easy Expression Test:
InOrder:
1 + 2 * 3
PostOrder:
1 2 + 3 *
PreOrder:
* + 1 2 3
Graphviz:
digraph G {
n5 [label="*"];
n3 [label="+"];
n1 [label="1"];
n3 -> n1;
n2 [label="2"];
n3 -> n2;
n5 -> n3;
n4 [label="3"];
n5 -> n4;
}
```

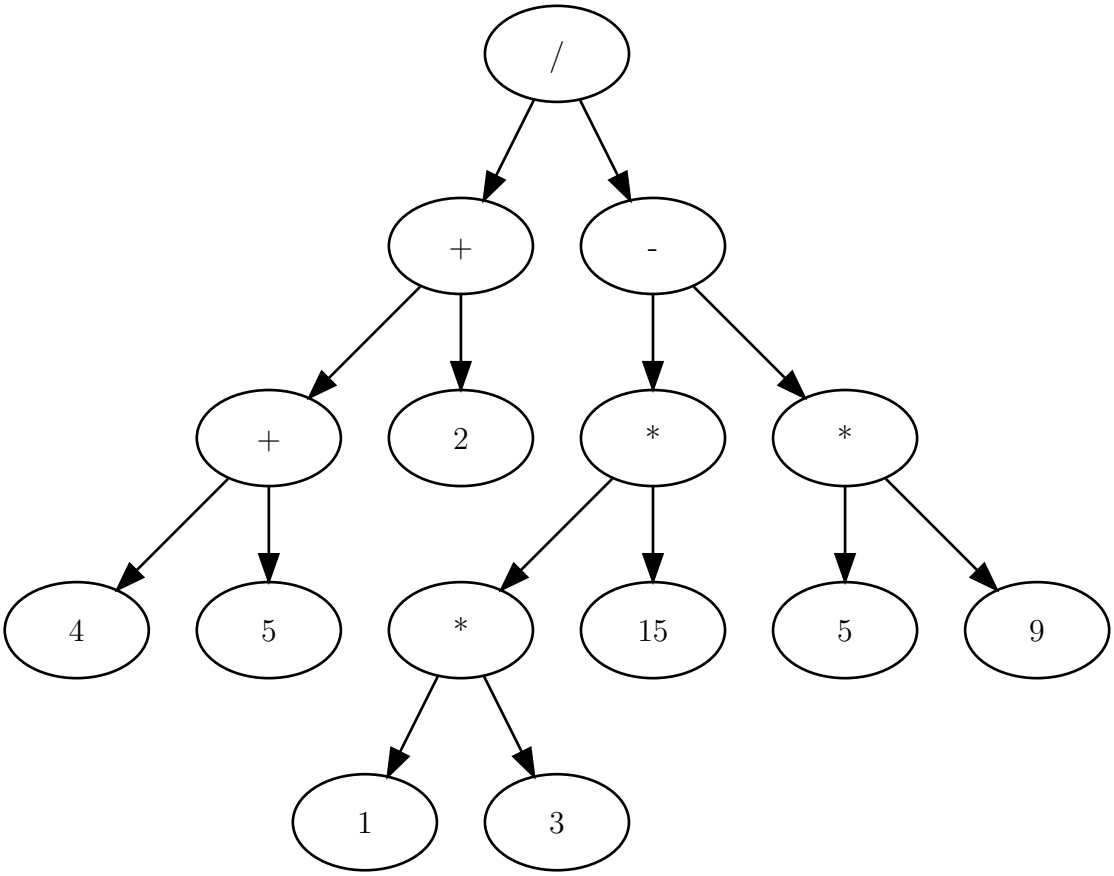
```
Result: 9
```

```
Difficult Expression Test:
InOrder:
50 / 2 * 25 - 14 * 1 + 14 + 14 * 1 - 25 + 14 + 20 + 2 * 22 - 23
PostOrder:
50 2 / 25 14 - 1 14 + * 14 1 25 - 14 + * + * 20 2 22 * + 23 - +
PreOrder:
+ * / 50 2 + * - 25 14 + 1 14 * 14 + - 1 25 14 - + 20 * 2 22 23
Graphviz:
digraph G {
n27 [label="+"];
n19 [label="*"];
n3 [label="/"];
n1 [label="50"];
n3 -> n1;
n2 [label="2"];
n3 -> n2;
n19 -> n3;
n18 [label="+"];
n10 [label="*"];
```

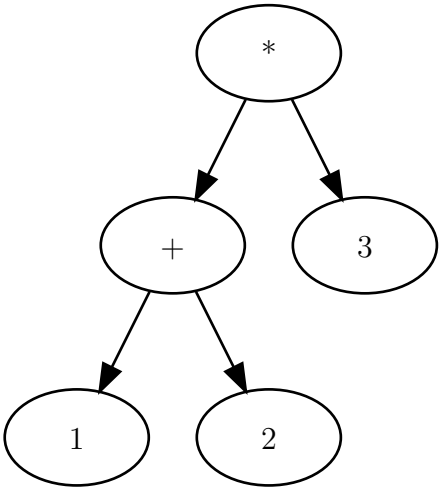
```
n6 [label="-"];
n4 [label="25"];
n6 -> n4;
n5 [label="14"];
n6 -> n5;
n10 -> n6;
n9 [label="+"];
n7 [label="1"];
n9 -> n7;
n8 [label="14"];
n9 -> n8;
n10 -> n9;
n18 -> n10;
n17 [label="*"];
n11 [label="14"];
n17 -> n11;
n16 [label="+"];
n14 [label="-"];
n12 [label="1"];
n14 -> n12;
n13 [label="25"];
n14 -> n13;
n16 -> n14;
n15 [label="14"];
n16 -> n15;
n17 -> n16;
n18 -> n17;
n19 -> n18;
n27 -> n19;
n26 [label="-"];
n24 [label="+"];
n20 [label="20"];
n24 -> n20;
n23 [label="*"];
n21 [label="2"];
n23 -> n21;
n22 [label="22"];
n23 -> n22;
n24 -> n23;
n26 -> n24;
n25 [label="23"];
n26 -> n25;
n27 -> n26;
}
```

Result: 666

2.5.2 divisionByZero Graph



2.5.3 easyExpression Graph



2.5.4 difficultExpression Graph