Abgabetermin: Mi, 03. 05. 2023

☒ **Gr. 1,** Dr. S. Wagner **Name** Elias Wurm **Aufwand in h** 1h 30min
☐ **Gr. 2,** Dr. D. Auer
☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter*in** _____ / _____

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
  { '+' Term     sem Write('+'); endsem
  | '-' Term     sem Write('-'); endsem
  } .
```

```
Term =
  Fact
  { '*' Fact     sem Write('*'); endsem
  | '/' Fact     sem Write('/'); endsem
  } .
```

```
Fact =
  number ↑n      sem Write(n); endsem
  | ident ↑id    sem Write(id); endsem
  | '(' Expr ')' .
```

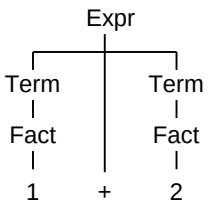
a) Entwickeln Sie eine ATG zur Transformation einfacher arithmetischer Ausdrücke von der Infix- in die Präfix-Notation, also z. B. von (*a* + *b*) * *c* nach * + *a b c*.

b) Implementieren Sie die ATG aus a) und testen Sie Ihre Implementierung ausführlich.

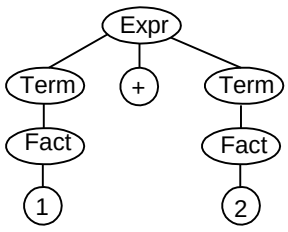
2. Syntaxbäume in kanonischer Form (4 + 6 + 4 Punkte)

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

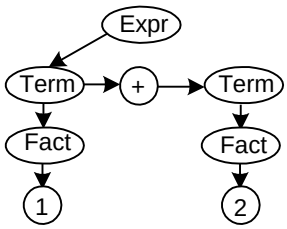
"Konventionelle" Darstellung



Baumdarstellung



Syntaxbäume sind somit Bäume, deren Knoten beliebig viele Nachfolgeknoten 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 einen Spezialfall von *Binärbäumen* 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 dann mit zwei Zeigern aus, 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:



(a) Entwickeln Sie aus der oben angegebenen Grammatik eine attributierte Grammatik (ATG), die für arithmetische Ausdrücke den Syntaxbaum in kanonischer Form aufbaut.

(b) Implementieren Sie diese attributierte Grammatik. Verwenden Sie dazu folgende Deklarationen:

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

(c) Für Testzwecke wollen Sie nun eine graphische Ausgabe der Syntaxbäume erzeugen, indem Sie *Graphviz* einsetzen (https://graphviz.org). *Graphviz* ist ein Werkzeug, das aus einer textuellen Beschreibung der Knoten und Kanten eines Graphen (oder eines Baumes) eine grafische Darstellung erzeugt. Beispielsweise erhält man mit der folgenden Beschreibung die Baumdarstellung für den Ausdruck 1 + 2.

```
digraph G {
  n1 [label="Expr"];
  n2 [label="Term"];
  n3 [label="Fact"];
  n4 [label="1"];
  n5 [label="+"];
  n6 [label="Term"];
  n7 [label="Fact"];
  n8 [label="2"];
  n1 -> n2;
  n2 -> n3;
  n3 -> n4;
  n1 -> n5;
  n1 -> n6;
  n6 -> n7;
  n7 -> n8;
}
```

Erweitern Sie den Datentyp `Node` um eine eindeutige ID für Knoten und implementieren Sie Prozeduren, um aus einem Syntaxbaum in kanonischer Form 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 Bilder ab.

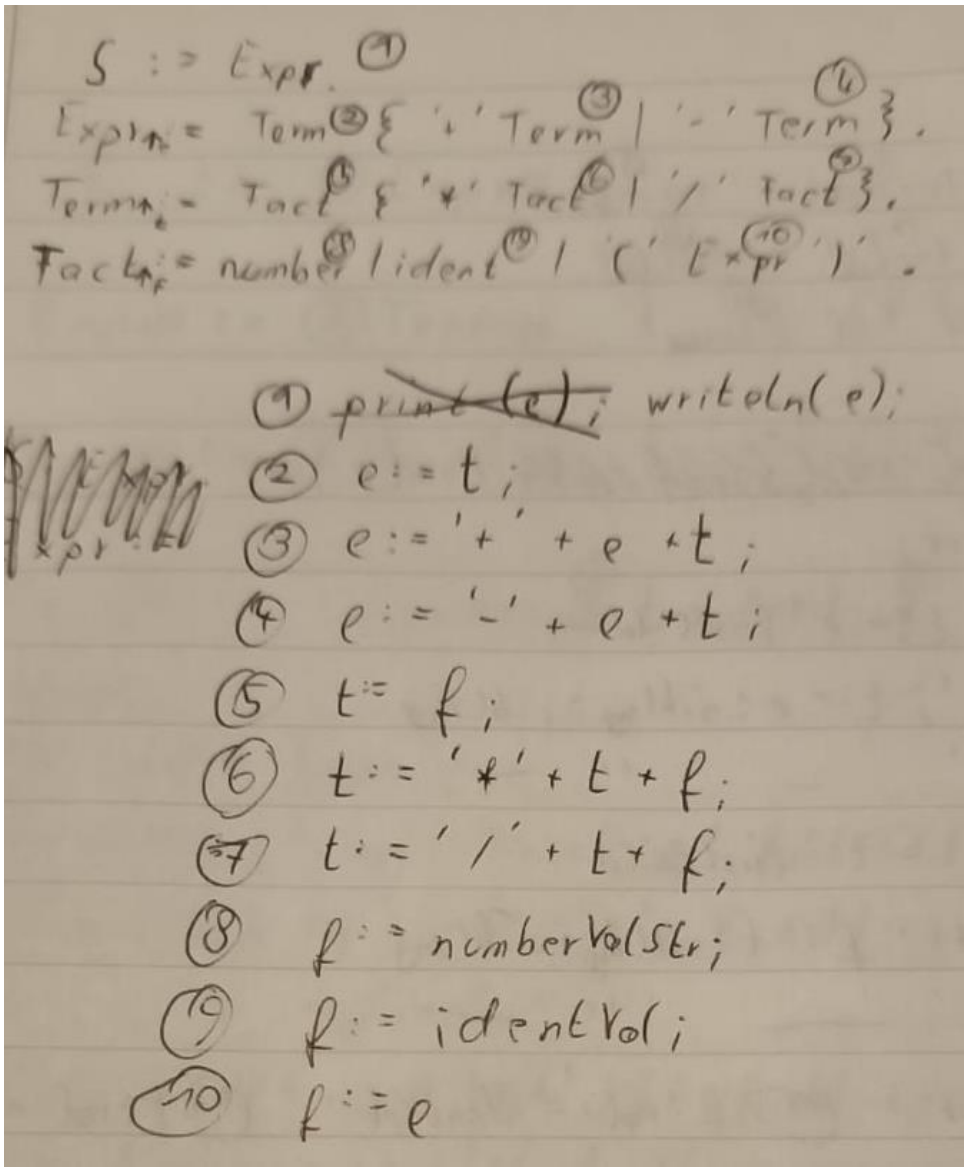
Das Werkzeug *Graphviz* können Sie lokal installieren (https://graphviz.org/) oder online verwenden, z.B. hier: https://dreampuf.github.io/.

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.

Aufgabe 1 - Transformation arithmetischer Ausdrücke

ATG:



Ich hoffe es ist einiger maßen leserlich :D

Zeitaufwand: ~30min

Code:

```
program InfixPrefix;
const
  eofCh = Chr(0);

type
  Symbol = (
```

```
    eofSy,
    errSy,
    plusSy, minusSy, timesSy, divSy,
    leftParSy, rightParSy,
    numberSy, identSy
    );

var
  line: string;         (* input sequence *)
  ch: char;             (* current character *)
  chNr: integer;        (* pos of ch *)
  sy: Symbol;           (* current symbol *)
  numberVal: integer;   (* numerical value if sy is a numberSy *)
  numberValStr: string; (* numerical value as string if sy is a numberSy *)
  identStr: string;     (* ident string value if sy is a identSy *)
  success: boolean;     (* syntax correct *)

(* SCANNER *)
procedure NewChar;
begin
  if(chNr < Length(line)) then
  begin
    Inc(chNr);
    ch := line[chNr];
  end else ch := eofCh;
end;

procedure NewSy;
begin
  while(ch = ' ') do NewChar;
  case ch of
    eofCh: sy := eofSy;
    '+':
    begin sy := plusSy; NewChar; end;
    '-':
    begin sy := minusSy; NewChar; end;
    '*':
    begin sy := timesSy; NewChar; end;
    '/':
    begin sy := divSy; NewChar; end;
    '(':
    begin sy := leftParSy; NewChar; end;
    ')':
    begin sy := rightParSy; NewChar; end;
    '0'..'9':
    begin
      sy := numberSy;
      numberval := 0;
      while((ch >= '0') and (ch <= '9')) do
      begin
        numberval := numberVal * 10 + Ord(ch) - Ord('0');
        NewChar;
      end;
      Str(numberVal, numberValStr);
```

```
    end;
    'a'..'z', 'A'..'Z', '_':
    begin
      sy := identSy;
      identStr := '';
      while((ch in ['a'..'z','A'..'Z','_','0'..'9'])) do
      begin
        identStr := identStr + ch;
        NewChar;
      end;
    end;
  else
    sy := errSy;
  end;
end;

(* Parser *)

procedure S; forward;
procedure Expr(var e: string); forward;
procedure Term(var t: string); forward;
procedure Fact(var f: string); forward;

procedure S;
var
  e: string;
begin
  success := true;
  Expr(e); if not success then exit;
  (* sem *) writeln(e); (* end sem *)
  if(sy <> eofSy) then
  begin
    success := false;
    exit;
  end;
end;

procedure Expr(var e: string);
var
  t: string;
begin
  Term(e); if not success then exit;
  while(sy = plusSy) or (sy = minusSy) do
    case sy of
      plusSy:
      begin
        NewSy;
        Term(t); if not success then exit;
        (* sem *) e := '+ ' + e + ' ' + t; (* end sem *)
      end;
      minusSy:
      begin
        NewSy;
        Term(t); if not success then exit;
```

```
        (* sem *) e := '- ' + e + ' ' + t; (* end sem *)
      end;
    end;
end;

procedure Term(var t: string);
var
  f: string;
begin
  Fact(t); if not success then exit;
  while(sy = timesSy) or (sy = divSy) do
    case sy of
      timesSy:
      begin
        NewSy;
        Fact(f); if not success then exit;
        (* sem *) t := '* ' + t + ' ' + f; (* end sem *)
      end;
      divSy:
      begin
        NewSy;
        Fact(f); if not success then exit;
        (* sem *) t := '/ ' + t + ' ' + f; (* end sem *)
      end;
    end;
end;

procedure Fact(var f: string);
begin
  case sy of
    numberSy:
    begin
      (* sem *) f := numberValStr; (* end sem *)
      NewSy;
    end;
    identSy:
    begin
      (* sem *) f := identStr; (* end sem *)
      NewSy;
    end;
    leftParSy:
    begin
      NewSy;
      Expr(f); if not success then exit;
      if(sy <> rightParSy) then
      begin success := false; Exit; end;
      NewSy;
    end;
  else
    success := false;
  end;
end;
```

```
(* Main *)
begin
  write('expr > '); readln(line);
  while(line <> '') do
  begin
    chNr := 0;
    NewChar;
    NewSy;
    S;
    if not success then writeln('syntax error');
    write('expr > '); readln(line);
  end;
end.
```

A lot of tests:

Test Case 1:
Input: 3+4*5/(6-2)
Expected Output: + 3 / * 4 5 - 6 2

```
expr > 3+4*5/(6-2)
+ 3 / * 4 5 - 6 2
```

Test Case 2:
Input: (2+3)*(4+5)
Expected Output: * + 2 3 + 4 5

```
expr > (2+3)*(4+5)
* + 2 3 + 4 5
```

Test Case 3:
Input: 2*(3+4)+5/6
Expected Output: + * 2 + 3 4 / 5 6

```
expr > 2*(3+4)+5/6
+ * 2 + 3 4 / 5 6
```

Test Case 4:
Input: 5 + ((1 + 2) * 4) - 3
Expected Output: - + 5 * + 1 2 4 3

```
expr > 5 + ((1 + 2) * 4) - 3
- + 5 * + 1 2 4 3
```

Test Case 5:
Input: 3 * (4 - 2) / (5 + 1)
Expected Output: / * 3 - 4 2 + 5 1

```
expr > 3 * (4 - 2) / (5 + 1)
/ * 3 - 4 2 + 5 1
```

Test Case 6:

Input: 1 + 2 + 3 + 4 + 5

Expected Output: + + + + 1 2 3 4 5

```
expr > 1 + 2 + 3 + 4 + 5
+ + + + 1 2 3 4 5
```

Test Case 7:

Input: (1 + 2) + (3 + 4) + 5

Expected Output: + + + 1 2 + 3 4 5

```
expr > (1 + 2) + (3 + 4) + 5
+ + + 1 2 + 3 4 5
```

Test Case 8:

Input: a + b * c - d / e

Expected Output: - + a * b c / d e

```
expr > a + b * c - d / e
- + a * b c / d e
```

Test Case 9:

Input: 32

Expected Output: 32

```
expr > 32
32
```

Test Case 10:

Input: a

Expected Output: a

```
expr > a
a
```

Test Case 11:

Input: (3)

Expected Output: 3

```
expr > (3)
3
```

Test Case 12:

Input: 2 * (3 +)

Expected Output: invalid

```
expr > 2 * (3 + )
syntax error
```

Test Case 13:
Input: 2 ++ 3
Expected Output: invalid

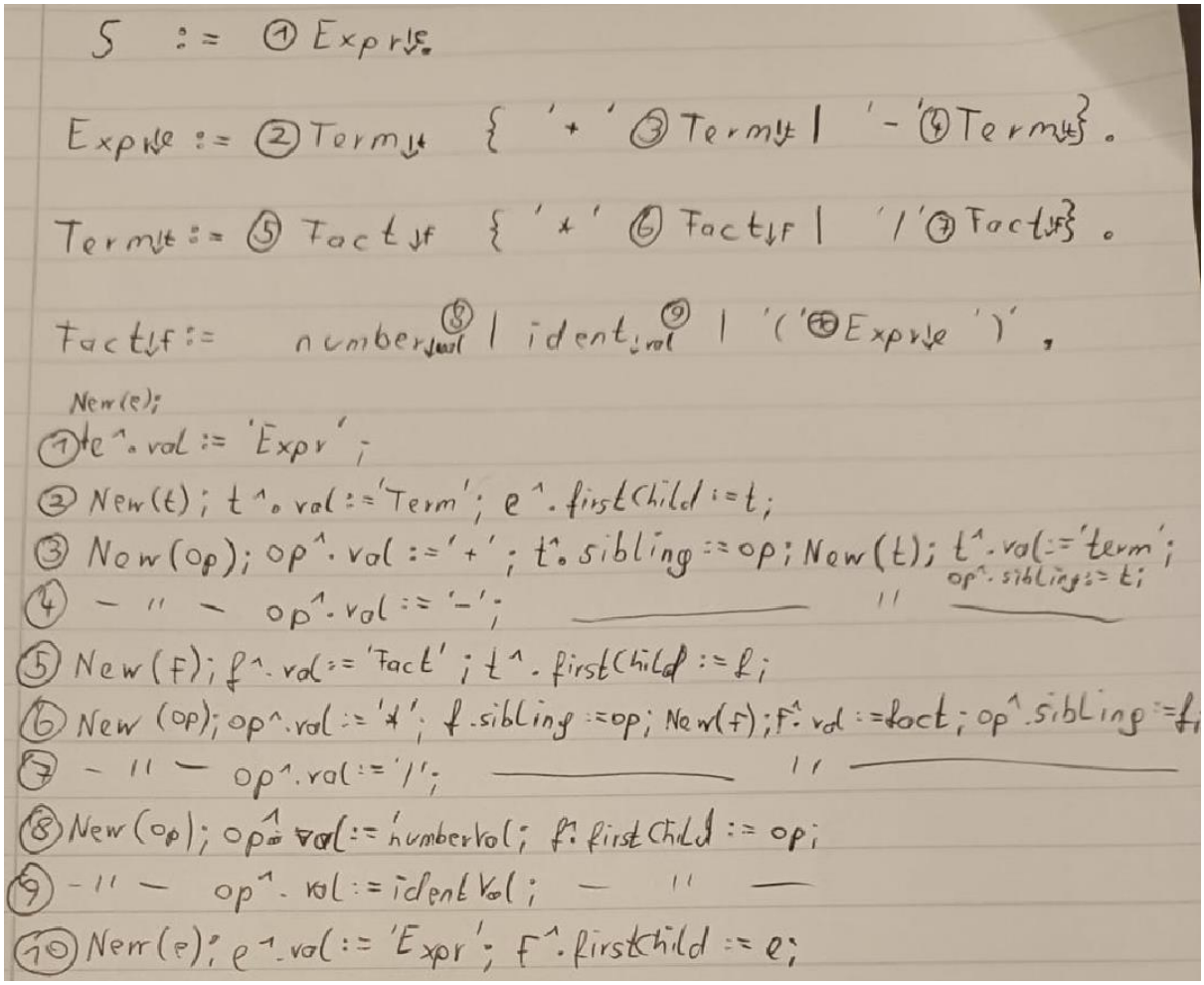
```
expr > 2 ++ 3
syntax error
```

Test Case 14:
Input: ()
Expected Output: invalid

```
expr > ()
syntax error
```

Aufgabe 2 - Syntaxbäume in kanonischer Form

ATG:



Ich hoffe es ist einiger maßen leserlich :D

Edit: mir ist noch aufgefallen das ich komplett vergessen hab das man Term + Term chainen kann (Bsp.: 1+2+3+4) und dann jeweils mehr Geschwister daran gehängt gehören (gleiche natürlich auch bei Fact).
Also stimmt meine Attribuierte Grammatik oben nicht ganz aber der code unten wurde angepasst.

Zeitaufwand: ~1h

Code:

```
program ExprSyntaxTree;

const
```

```
  eofCh = Chr(0);

type
  Symbol = (
    eofSy,
    errSy,
    plusSy, minusSy, timesSy, divSy,
    leftParSy, rightParSy,
    numberSy, identSy
    );
  NodePtr = ^Node;
  Node = record
    id: string; (* id of node, used for graphical representation in graphviz
*)
    firstChild, sibling: NodePtr;
    val: string; (* nonterminal, operator or operand as string *)
  end;
  TreePtr = NodePtr;

var
  line: string;         (* input sequence *)
  ch: char;             (* current character *)
  chNr: integer;        (* pos of ch *)
  sy: Symbol;           (* current symbol *)
  numberVal: integer;   (* numerical value if sy is a numberSy *)
  numberValStr: string; (* numerical value as string if sy is a numberSy *)
  identStr: string;     (* ident string value if sy is a identSy *)
  success: boolean;     (* syntax correct *)
  idCounter: integer;   (* for graphix representation ids for nodes are
needed,
                           here an incremental number is used as id *)

(* SCANNER *)
procedure NewChar;
begin
  if(chNr < Length(line)) then
  begin
    Inc(chNr);
    ch := line[chNr];
  end else ch := eofCh;
end;

procedure NewSy;
begin
  while(ch = ' ') do NewChar;
  case ch of
    eofCh: sy := eofSy;
    '+':
```

```
    begin sy := plusSy; NewChar; end;
    '-':
    begin sy := minusSy; NewChar; end;
    '*':
    begin sy := timesSy; NewChar; end;
    '/':
    begin sy := divSy; NewChar; end;
    '(':
    begin sy := leftParSy; NewChar; end;
    ')':
    begin sy := rightParSy; NewChar; end;
    '0'..'9':
    begin
      sy := numberSy;
      numberval := 0;
      while((ch >= '0') and (ch <= '9')) do
      begin
        numberval := numberVal * 10 + Ord(ch) - Ord('0');
        NewChar;
      end;
      Str(numberVal, numberValStr);
    end;
    'a'..'z', 'A'..'Z', '_':
    begin
      sy := identSy;
      identStr := '';
      while((ch in ['a'..'z','A'..'Z','_','0'..'9'])) do
      begin
        identStr := identStr + ch;
        NewChar;
      end;
    end;
  else
    sy := errSy;
  end;
end;

(* Helper functions for parser *)

function NewNode(val: string): NodePtr;
var
  n: NodePtr;
  id: string;
begin
  New(n);
  Str(idCounter, id);
  Inc(idCounter);
  n^.id := 'n' + id;
```

```
  n^.val := val;
  n^.firstChild := nil;
  n^.sibling := nil;
  NewNode := n;
end;

function AddNewSibling(var node: NodePtr; newNodeVal: string): NodePtr;
var
  newSibling: NodePtr;
begin
  newSibling := NewNode(newNodeVal);
  node^.sibling := newSibling;
  AddNewSibling := newSibling;
end;

procedure DisposeTree(var t: TreePtr);
begin
  if t <> nil then
  begin
    DisposeTree(t^.firstChild);
    DisposeTree(t^.sibling);
    Dispose(t);
  end;
end;

procedure PrintTree(node: TreePtr);

  procedure PrintNodes(n: NodePtr);
  begin
    if(n <> nil) then
    begin
      WriteLn(n^.id, ' [label="', n^.val, '"];');
      PrintNodes(n^.sibling);
      PrintNodes(n^.firstChild);
    end;
  end;

  procedure PrintRelations(n: NodePtr);
  begin
    if(n <> nil) then
    begin
      if(n^.firstChild <> nil) then WriteLn(n^.id, ' -> ', n^.firstChild^.id,
' [label="firstChild"];');
      if(n^.sibling <> nil) then WriteLn(n^.id, ' -> ', n^.sibling^.id, '
[label="sibling"];');
      PrintRelations(n^.firstChild);
      PrintRelations(n^.sibling);
    end;
```

```
  end;

begin
  WriteLn('digraph G {');
  PrintNodes(node);
  PrintRelations(node);
  WriteLn('}');
end;

(* Parser *)

procedure S; forward;
procedure Expr(var e: NodePtr); forward;
procedure Term(var t: NodePtr); forward;
procedure Fact(var f: NodePtr); forward;

procedure S;
var
  t: NodePtr;
begin
  success := true;
  (* sem *) idCounter := 0; t := NewNode('Expr'); (* end sem *)
  Expr(t); if not success then exit;
  if(sy <> eofSy) then
  begin
    success := false;
    exit;
  end;
  (* sem *) PrintTree(t); DisposeTree(t); (* end sem *)
end;

procedure Expr(var e: NodePtr);
var
  curSibling: NodePtr;
begin
  (* sem *) curSibling := NewNode('Term'); e^.firstChild := curSibling; (* end
sem *)
  Term(curSibling); if not success then exit;
  while(sy = plusSy) or (sy = minusSy) do
    case sy of
      plusSy:
      begin
        NewSy;
        (* sem *)
        curSibling := AddNewSibling(curSibling, '+');
        curSibling := AddNewSibling(curSibling, 'Term');
        (* end sem *)
        Term(curSibling); if not success then exit;
```

```
      end;
      minusSy:
      begin
        NewSy;
        (* sem *)
        curSibling := AddNewSibling(curSibling, '-');
        curSibling := AddNewSibling(curSibling, 'Term');
        (* end sem *)
        Term(curSibling); if not success then exit;
      end;
    end;
end;

procedure Term(var t: NodePtr);
var
  curSibling: NodePtr;
begin
  (* sem *) curSibling := NewNode('Fact'); t^.firstChild := curSibling; (* end
sem *)
  Fact(t^.firstChild); if not success then exit;
  while(sy = timesSy) or (sy = divSy) do
    case sy of
      timesSy:
      begin
        NewSy;
        (* sem *)
        curSibling := AddNewSibling(curSibling, '*');
        curSibling := AddNewSibling(curSibling, 'Fact');
        (* end sem *)
        Fact(curSibling); if not success then exit;
      end;
      divSy:
      begin
        NewSy;
        (* sem *)
        curSibling := AddNewSibling(curSibling, '/');
        curSibling := AddNewSibling(curSibling, 'Fact');
        (* end sem *)
        Fact(curSibling); if not success then exit;
      end;
    end;
end;

procedure Fact(var f: NodePtr);
begin
  case sy of
    numberSy:
    begin
```

```
      (* sem *) f^.firstChild := NewNode(numberValStr); (* end sem *)
      NewSy;
    end;
    identSy:
    begin
      (* sem *) f^.firstChild := NewNode(identStr); (* end sem *)
      NewSy;
    end;
    leftParSy:
    begin
      NewSy;
      (* sem *) f^.firstChild := NewNode('Expr'); (* end sem *)
      Expr(f^.firstChild); if not success then exit;
      if(sy <> rightParSy) then
      begin success := false; Exit; end;
      NewSy;
    end;
  else
    success := false;
  end;
end;

(* Main *)
begin
  write('expr > '); readln(line);
  while(line <> '') do
  begin
    chNr := 0;
    NewChar;
    NewSy;
    S;
    if not success then writeln('syntax error');
    write('expr > '); readln(line);
  end;
end.
```

Test:

```
expr > 4*(3+d)
digraph G {
n0 [label="Expr"];
n1 [label="Term"];
n2 [label="Fact"];
n4 [label="*"];
n5 [label="Fact"];
n6 [label="Expr"];
n7 [label="Term"];
n10 [label="+"];
n11 [label="Term"];
n12 [label="Fact"];
n13 [label="d"];
n8 [label="Fact"];
n9 [label="3"];
n3 [label="4"];
n0 -> n1 [label="firstChild"];
n1 -> n2 [label="firstChild"];
n2 -> n3 [label="firstChild"];
n2 -> n4 [label="sibling"];
n4 -> n5 [label="sibling"];
n5 -> n6 [label="firstChild"];
n6 -> n7 [label="firstChild"];
n7 -> n8 [label="firstChild"];
n7 -> n10 [label="sibling"];
n8 -> n9 [label="firstChild"];
n10 -> n11 [label="sibling"];
n11 -> n12 [label="firstChild"];
n12 -> n13 [label="firstChild"];
}
```