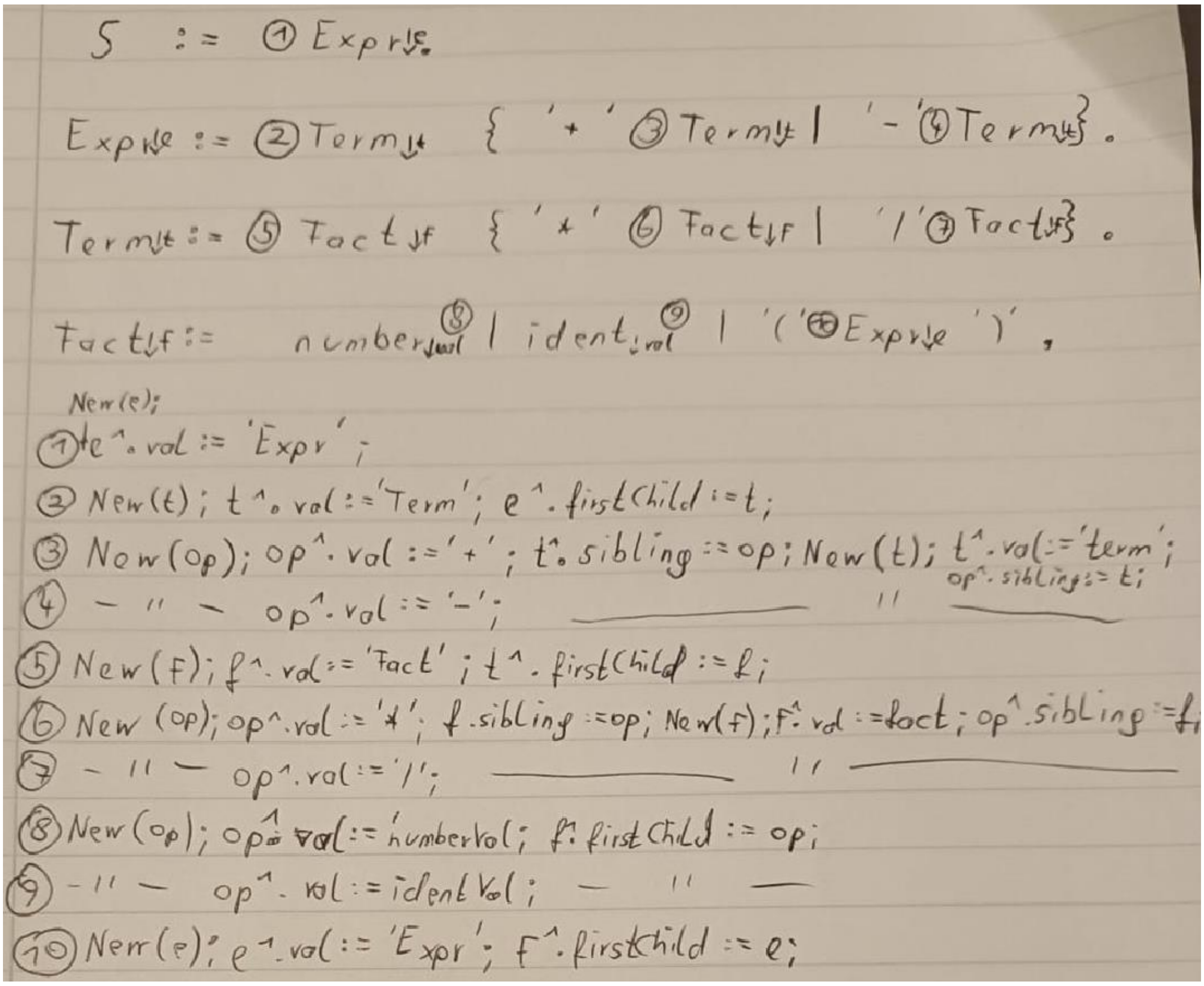
Aufgabe 2 - Syntaxbäume in kanonischer Form

ATG:



```
S     ::= ① Expr↑e .

Expr↑e ::= ② Term↓t   { '+' ③ Term↓t | '-' ④ Term↓t } .

Term↓t ::= ⑤ Fact↓f   { '*' ⑥ Fact↓f | '/' ⑦ Fact↓f } .

Fact↓f ::=   number↓val ⑧ | ident↓val ⑨ | '(' ⑩ Expr↓e ')' ,

New(e);
① e^.val := 'Expr';
② New(t); t^.val:='Term'; e^.firstChild:=t;
③ New(op); op^.val :='+'; t^.sibling :=op; New(t); t^.val:='term';
                                                  op^.sibling:= t;
④ — '' — op^.val := '-'; ————————— '' —————
⑤ New(f); f^.val:= 'Fact'; t^.firstChild :=f;
⑥ New (op); op^.val := '*'; f.sibling :=op; New(f); f^.val :=fact; op^.sibling :=f
⑦ — '' — op^.val:='/'; ————————— '' —————————
⑧ New(op); op^.val:= numberVal; f^.firstChild := op;
⑨ — '' — op^.val:=identVal; — '' —
⑩ New(e); e^.val:= 'Expr'; f^.firstchild := e;
```

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

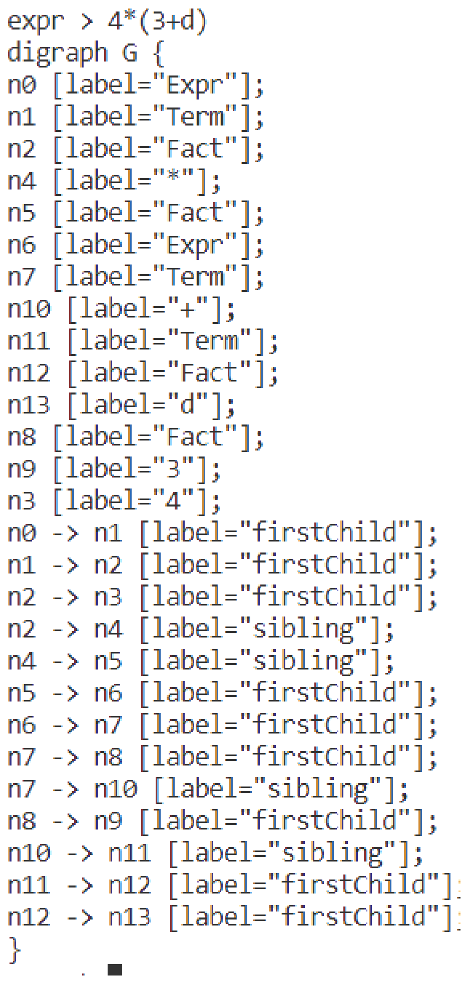
Code:

```
program ExprSyntaxTree;

const
```

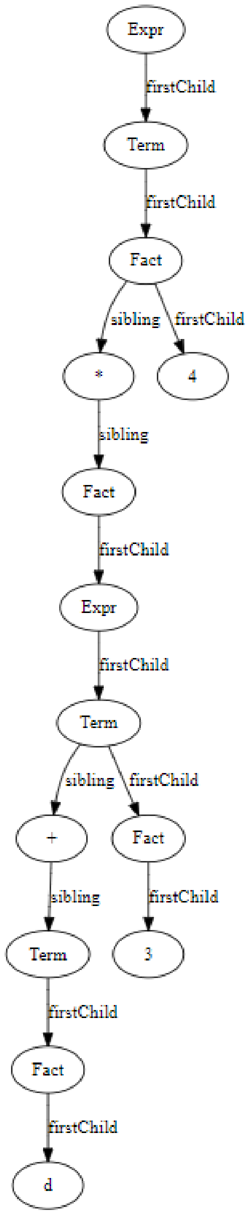
```
  eofCh = Chr(0);

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"]
}
```


Expr
firstChild
Term
firstChild
Fact
sibling
firstChild
*
4
sibling
Fact
firstChild
Expr
firstChild
Term
sibling
firstChild
+
Fact
sibling
firstChild
Term
3
firstChild
Fact
firstChild
d