Aufgabe 2 - Syntaxbäume in kanonischer Form

**ATG:**

Text, letter

Description automatically generated

*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:**

